目录

前言

Java

Android

数据结构

算法

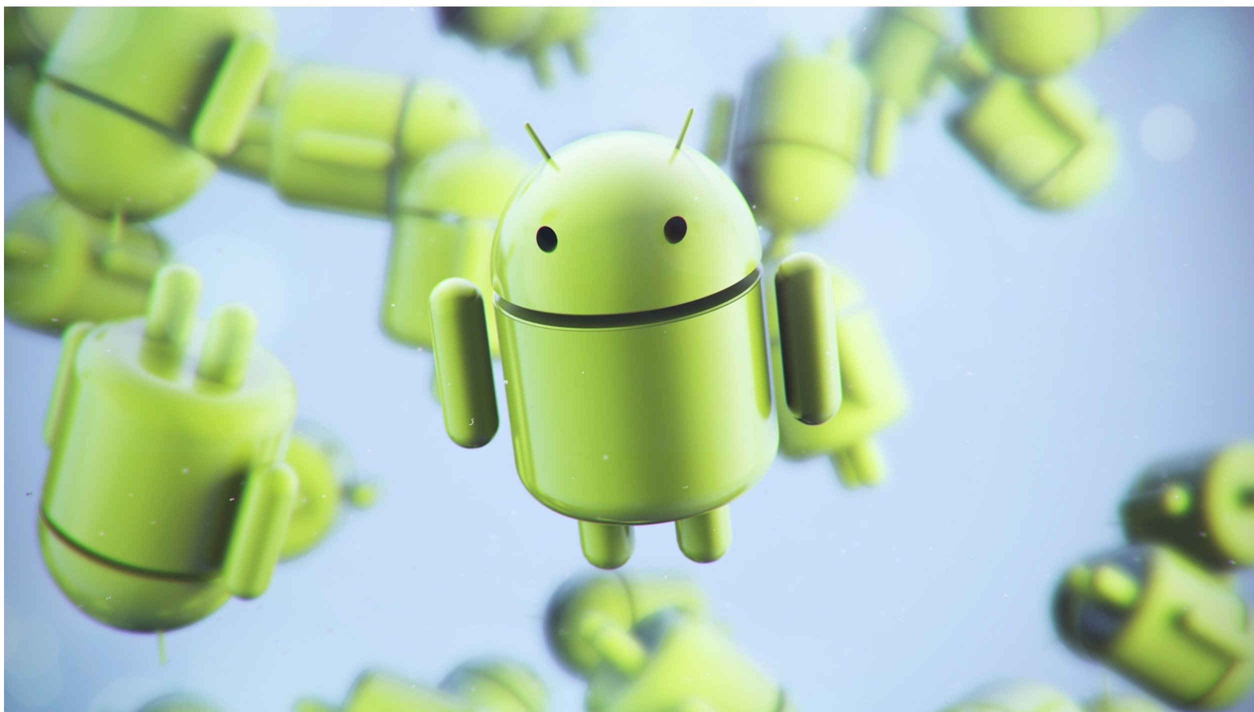
设计模式

计算机网络

操作系统

数据库

Android校招面试指南



Java

- Java基础
- Java并发
- Java虚拟机

Android

- Android基础
- Android进阶
- 开源框架

数据结构

- 线性表
- 栈和队
- 树

- 图
- 散列查找
- 排序
- 海量数据处理

算法

- 剑指offer
- LeetCode

设计模式

- 创建型模式
- 结构型模式
- 行为型模式

计算机网络

- TCP/IP
- HTTP
- HTTPS

操作系统

- 概述
- 进程与线程
- 内存管理

数据库

- SQL语句

致谢

贡献者	贡献内容	贡献者	贡献内容
YiKun	Java集合	Zane	AbstractQueuedSynchronizer
DERRANTCM	剑指offer	占小狼	ConcurrentHashMap
skywang12345	数据结构	IAM四十二	Android动画总结
Carson_Ho	Android基础	me115	图解设计模式
Piasy	Android开源框架	朱祁林	https原理解析
stormzhang	Android全局异常	Trinea	Parcelable和Serializable

贡献者	贡献内容	贡献者	贡献内容
AriaLyy	多线程断点续传	JackieYeah	Java深拷贝和浅拷贝
ZHANG_L	Android进程优先级	尹star	Android Context详解
HELLO丶GUY	Fragment详解	Shawon	Android推送技术
徐凯强Andy	动态规划	aaronice	LeetCode/LintCode题解
码农一枚	BlockingQueue	Alexia	Java transien和finally return
朔野	Android Apk安装过程	黑泥卡	Dialog和PopupWindow

持续更新，仍有更多内容尚未完善，欢迎大家投稿。

一、前言

本部分内容主要包含以下：

- Java集合
- Java反射
- Java注解
- Java反射
- Java IO
- 其他面试点

以上内容都是Java中的基础知识，对于Java的学习很有帮助。其中集合、反射、IO等都是面试常问知识点，是必须要掌握的，因此整理在本部分当中。

二、目录

一、Java集合类简介：

Java集合大致可以分为Set、List、Queue和Map四种体系。

其中Set代表无序、不可重复的集合；List代表有序、重复的集合；而Map则代表具有映射关系的集合。Java 5 又增加了Queue体系集合，代表一种队列集合实现。

Java集合就像一种容器，可以把多个对象（实际上是对象的引用，但习惯上都称对象）“丢进”该容器中。从Java 5 增加了泛型以后，Java集合可以记住容器中对象的数据类型，使得编码更加简洁、健壮。

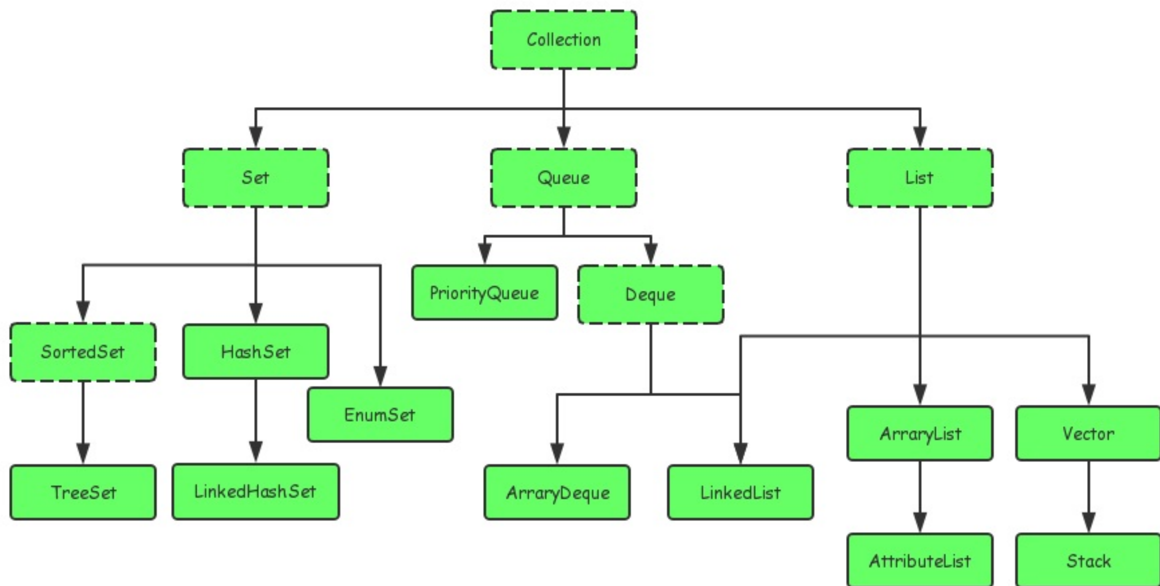
1.Java集合和数组的区别：

①.数组长度在初始化时指定，意味着只能保存定长的数据。而集合可以保存数量不确定的数据。同时可以保存具有映射关系的数据（即关联数组，键值对 key-value）。

②.数组元素即可以是基本类型的值，也可以是对象。集合里只能保存对象（实际上只是保存对象的引用变量），基本数据类型的变量要转换成对应的包装类才能放入集合类中。

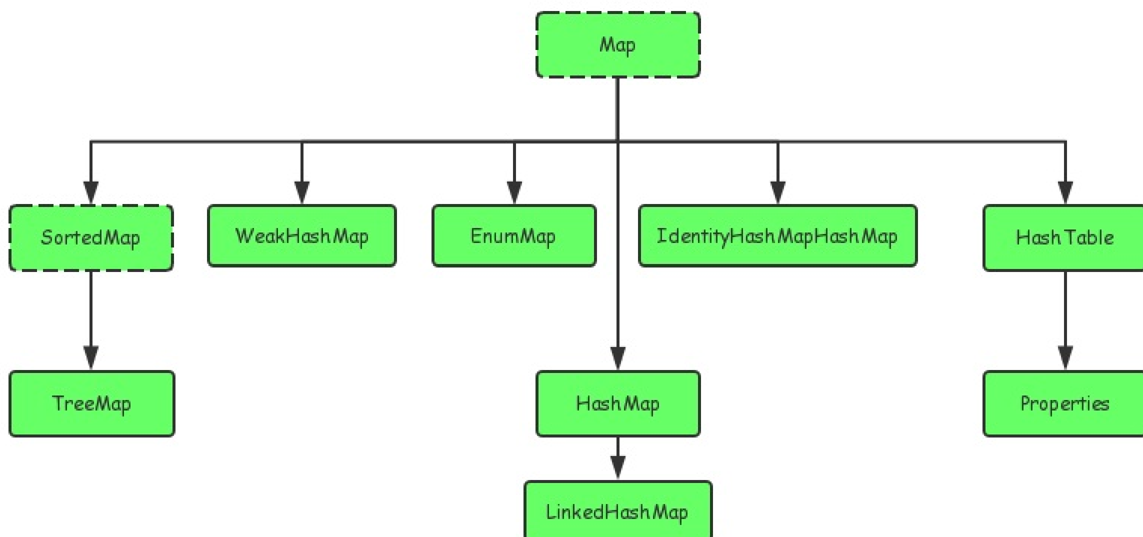
2.Java集合类之间的继承关系:

Java的集合类主要由两个接口派生而出：Collection和Map,Collection和Map是Java集合框架的根接口。



图中，ArrayList,HashSet,LinkedList,TreeSet是我们经常会有用到的已实现的集合类。

Map实现类用于保存具有映射关系的数据。Map保存的每项数据都是key-value对，也就是由key和value两个值组成。Map里的key是不可重复的，key用户标识集合里的每项数据。



图中，HashMap，TreeMap是我们经常会用到的集合类。

二、Collection接口：

1.简介

Collection接口是Set,Queue,List的父接口。Collection接口中定义了多种方法可供其子类进行实现，以实现数据操作。由于方法比较多，就偷个懒，直接把JDK文档上的内容搬过来。

1.1.接口中定义的方法

方法摘要	
boolean	add(E e) 确保此 collection 包含指定的元素（可选操作）。
boolean	addAll(Collection<? extends E> c) 将指定 collection 中的所有元素都添加到此 collection 中（可选操作）。
void	clear() 移除此 collection 中的所有元素（可选操作）。
boolean	contains(Object o) 如果此 collection 包含指定的元素，则返回 true。
boolean	containsAll(Collection<?> c) 如果此 collection 包含指定 collection 中的所有元素，则返回 true。
boolean	equals(Object o) 比较此 collection 与指定对象是否相等。
int	hashCode() 返回此 collection 的哈希码值。
boolean	isEmpty() 如果此 collection 不包含元素，则返回 true。
Iterator<E>	iterator() 返回在此 collection 的元素上进行迭代的迭代器。
boolean	remove(Object o) 从此 collection 中移除指定元素的单个实例，如果存在的话（可选操作）。
boolean	removeAll(Collection<?> c) 移除此 collection 中那些也包含在指定 collection 中的所有元素（可选操作）。
boolean	retainAll(Collection<?> c) 仅保留此 collection 中那些也包含在指定 collection 的元素（可选操作）。
int	size() 返回此 collection 中的元素数。
Object[]	toArray() 返回包含此 collection 中所有元素的数组。
<T> T[]	toArray(T[] a) 返回包含此 collection 中所有元素的数组；返回数组的运行时类型与指定数组的运行时类型相同。

可以看出Collection用法有：添加元素，删除元素，返回Collection集合的个数以及清空集合等。 其中重点介绍iterator()方法，该方法的返回值是Iterator。

1.2.使用Iterator遍历集合元素

Iterator接口经常被称作迭代器，它是Collection接口的父接口。但Iterator主要用于遍历集合中的元素。 Iterator接口中主要定义了2个方法：

boolean	hasNext() 如果仍有元素可以迭代，则返回 true。
E	next() 返回迭代的下一个元素。

下面程序简单示范了通过Iterator对象逐个获取元素的逻辑。

```
public class IteratorExample {
    public static void main(String[] args){
        //创建集合，添加元素
        Collection<Day> days = new ArrayList<Day>();
        for(int i =0;i<10;i++){
            Day day = new Day(i,i*60,i*3600);
            days.add(day);
        }
        //获取days集合的迭代器
        Iterator<Day> iterator = days.iterator();
        while(iterator.hasNext()){//判断是否有下一个元素
            Day next = iterator.next();//取出该元素
            //逐个遍历，取得元素后进行后续操作
            .....
        }
    }

}
```

注意：当使用Iterator对集合元素进行迭代时，把集合元素的值传给了迭代变量（就如同参数传递是值传递，基本数据类型传递的是值，引用类型传递的仅仅是对象的引用变量。

下面的程序演示了这一点：

```
public class IteratorExample {
  public static void main(String[] args) {
            List<MyObject> list = new ArrayList<>();
            for (int i = 0; i < 10; i++) {
                list.add(new MyObject(i));
            }

            System.out.println(list.toString());

            Iterator<MyObject> iterator = list.iterator();//集合元素的值传给了迭代变量，仅仅传递了对象引用。保存的仅仅是指向对象内存空间的地址

            while (iterator.hasNext()) {
                MyObject next = iterator.next();
                next.num = 99;
            }

            System.out.println(list.toString());
    }
    static class MyObject {
        int num;

        MyObject(int num) {
            this.num = num;
        }

        @Override
        public String toString() {
            return String.valueOf(num);
        }
    }
}
```

输出结果如下：

> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>
> [99, 99, 99, 99, 99, 99, 99, 99, 99, 99]

下面具体介绍Collection接口的三个子接口Set，List，Queue。

2.Set集合

简介

Set集合与Collection集合基本相同，没有提供任何额外的方法。实际上Set就是Collection，只是行为略有不同（Set不允许包含重复元素）。

Set集合不允许包含相同的元素，如果试图把两个相同的元素加入同一个Set集合中，则添加操作失败，add()方法返回false，且新元素不会被加入。

3.List集合

3.1.简介

List集合代表一个元素有序、可重复的集合，集合中每个元素都有其对应的顺序索引。List集合允许使用重复元素，可以通过索引来访问指定位置的集合元素 。List集合默认按元素的添加顺序设置元素的索引，例如第一个添加的元素索引为0，第二个添加的元素索引为1......

List作为Collection接口的子接口，可以使用Collection接口里的全部方法。而且由于List是有序集合，因此List集合里增加了一些根据索引来操作集合元素的方法。

3.2.接口中定义的方法

> **void add(int index, Object element):** 在列表的指定位置插入指定元素（可选操作）。
>
> **boolean addAll(int index, Collection<? extends E> c) :** 将集合c 中的所有元素都插入到列表中的指定位置index处。
>
> **Object get(index):** 返回列表中指定位置的元素。
>
> **int indexOf(Object o):** 返回此列表中第一次出现的指定元素的索引；如果此列表不包含该元素，则返回 -1。
>
> **int lastIndexOf(Object o):** 返回此列表中最后出现的指定元素的索引；如果列表不包含此元素，则返回 -1。
>
> **Object remove(int index):** 移除列表中指定位置的元素。
>
> **Object set(int index, Object element):** 用指定元素替换列表中指定位置的元素。
>
> **List subList(int fromIndex, int toIndex):** 返回列表中指定的 fromIndex（包括 ）和 toIndex（不包括）之间的所有集合元素组成的子集。
>
> **Object[] toArray():** 返回按适当顺序包含列表中的所有元素的数组（从第一个元素到最后一个元素）。

除此之外，Java 8还为List接口添加了如下两个默认方法。

> **void replaceAll(UnaryOperator operator):** 根据operator指定的计算规则重新设置List集合的所有元素。
>
> **void sort(Comparator c):** 根据Comparator参数对List集合的元素排序。

4.Queue集合

4.1.简介

Queue用户模拟队列这种数据结构，队列通常是指“先进先出”(FIFO，first-in-first-out)的容器。队列的头部是在队列中存放时间最长的元素，队列的尾部是保存在队列中存放时间最短的元素。新元素插入（offer）到队列的尾部，访问元素（poll）操作会返回队列头部的元素。通常，队列不允许随机访问队列中的元素。

4.2.接口中定义的方法

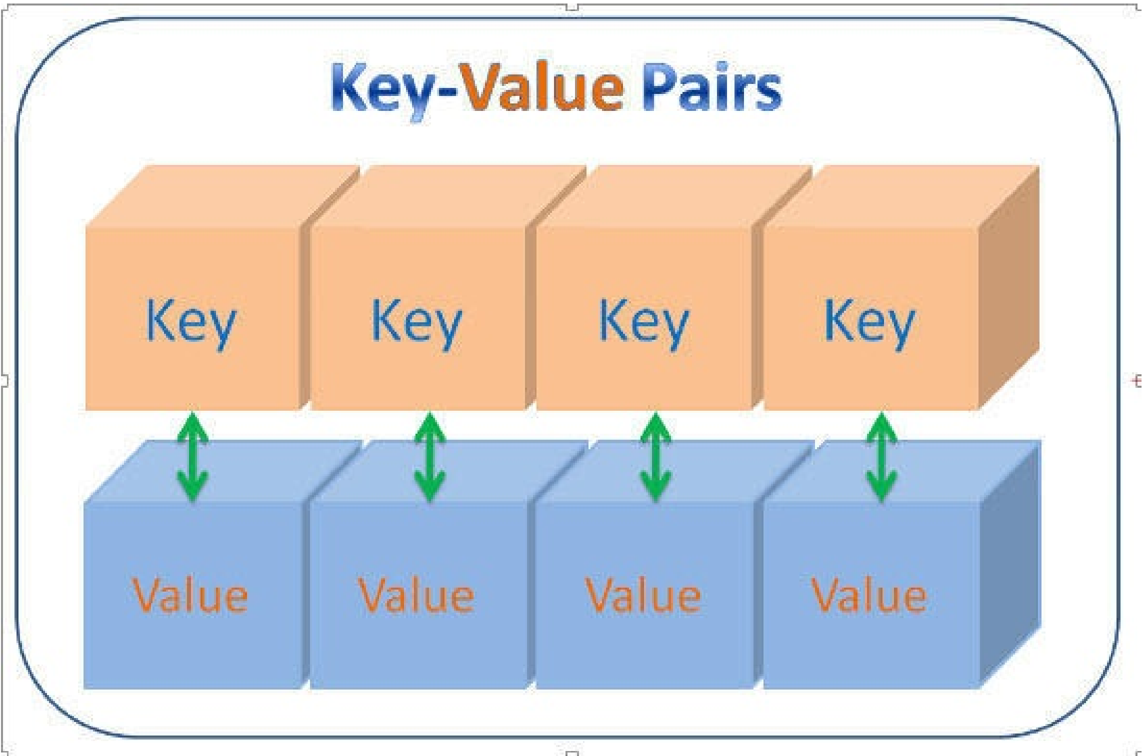
boolean	add(E e) 将指定的元素插入此队列（如果立即可行且不会违反容量限制），在成功时返回 true，如果当前没有可用的空间，则抛出 IllegalStateException。
E	element() 获取，但是不移除此队列的头。
boolean	offer(E e) 将指定的元素插入此队列（如果立即可行且不会违反容量限制），当使用有容量限制的队列时，此方法通常要优于 add(E)，后者可能无法插入元素，而只是抛出一个异常。
E	peek() 获取但不移除此队列的头；如果此队列为空，则返回 null。
E	poll() 获取并移除此队列的头，如果此队列为空，则返回 null。
E	remove() 获取并移除此队列的头。

三、Map集合

1.简介

Map用户保存具有映射关系的数据，因此Map集合里保存着两组数，一组值用户保存Map里的key,另一组值用户保存Map里的value，key和value都可以是任何引用类型的数据。Map的key不允许重复，即同一个Map对象的任何两个key通过equals方法比较总是返回false。

如下图所描述，key和value之间存在单向一对一关系，即通过指定的key,总能找到唯一的、确定的value。从Map中取出数据时，只要给出指定的key，就可以取出对应的value。



2.Map集合与Set集合、List集合的关系

①.与Set集合的关系

如果 把Map里的所有key放在一起看，它们就组成了一个Set集合（所有的key没有顺序，key与key之间不能重复），实际上Map确实包含了一个keySet()方法，用户返回Map里所有key组成的Set集合。

②.与List集合的关系

如果把Map里的所有value放在一起来看，它们又非常类似于一个List：元素与元素之间可以重复，每个元素可以根据索引来查找，只是Map中索引不再使用整数值，而是以另外一个对象作为索引。

3.接口中定义的方法

void	clear() 从此映射中移除所有映射关系（可选操作）。
boolean	containsKey(Object key) 如果此映射包含指定键的映射关系，则返回 true。
boolean	containsValue(Object value) 如果此映射将一个或多个键映射到指定值，则返回 true。
Set<Map.Entry<K,V>>	entrySet() 返回此映射中包含的映射关系的 Set 视图。
boolean	equals(Object o) 比较指定的对象与此映射是否相等。
V	get(Object key) 返回指定键所映射的值；如果此映射不包含该键的映射关系，则返回 null。
int	hashCode() 返回此映射的哈希码值。
boolean	isEmpty() 如果此映射未包含键-值映射关系，则返回 true。
Set<K>	keySet() 返回此映射中包含的键的 Set 视图。
V	put(K key, V value) 将指定的值与此映射中的指定键关联（可选操作）。
void	putAll(Map<? extends K,? extends V> m) 从指定映射中将所有映射关系复制到此映射中（可选操作）。
V	remove(Object key) 如果存在一个键的映射关系，则将其从此映射中移除（可选操作）。
int	size() 返回此映射中的键-值映射关系数。
Collection<V>	values() 返回此映射中包含的值的 Collection 视图。

Map中还包括一个内部类Entry，该类封装了一个key-value对。Entry包含如下三个方法：

boolean	equals(Object o) 比较指定对象与此项的相等性。
K	getKey() 返回与此项对应的键。
V	getValue() 返回与此项对应的值。
int	hashCode() 返回此映射项的哈希码值。
V	setValue(V value) 用指定的值替换与此项对应的值（可选操作）。

Map集合最典型的用法就是成对地添加、删除key-value对，然后就是判断该Map中是否包含指定key，是否包含指定value，也可以通过Map提供的keySet()方法获取所有key组成的集合，然后使用foreach循环来遍历Map的所有key，根据key即可遍历所有的value。下面程序代码示范Map的一些基本功能：

```
public class MapTest {
    public static void main(String[] args){
        Day day1 = new Day(1, 2, 3);
        Day day2 = new Day(2, 3, 4);
        Map<String,Day> map = new HashMap<String,Day>();
        //成对放入key-value对
        map.put("第一个", day1);
        map.put("第二个", day2);
        //判断是否包含指定的key
        System.out.println(map.containsKey("第一个"));
        //判断是否包含指定的value
        System.out.println(map.containsValue(day1));
        //循环遍历
        //1.获得Map中所有key组成的set集合
        Set<String> keySet = map.keySet();
        //2.使用foreach进行遍历
        for (String key : keySet) {
            //根据key获得指定的value
            System.out.println(map.get(key));
        }
        //根据key来移除key-value对
        map.remove("第一个");
        System.out.println(map);
    }

}
```

输出结果：

> true
>
> true
>
> Day [hour=2, minute=3, second=4]
>
> Day [hour=1, minute=2, second=3]
>
> {第二个=Day [hour=2, minute=3, second=4]}

一、概述

> 以数组实现。节约空间，但数组有容量限制。超出限制时会增加50%容量，用System.arraycopy()复制到新的数组，因此最好能给出数组大小的预估值。默认第一次插入元素时创建大小为10的数组。
>
> 按数组下标访问元素—get(i)/set(i,e) 的性能很高，这是数组的基本优势。
>
> 直接在数组末尾加入元素—add(e)的性能也高，但如果按下标插入、删除元素—add(i,e), remove(i), remove(e)，则要用System.arraycopy()来移动部分受影响的元素，性能就变差了，这是基本劣势。

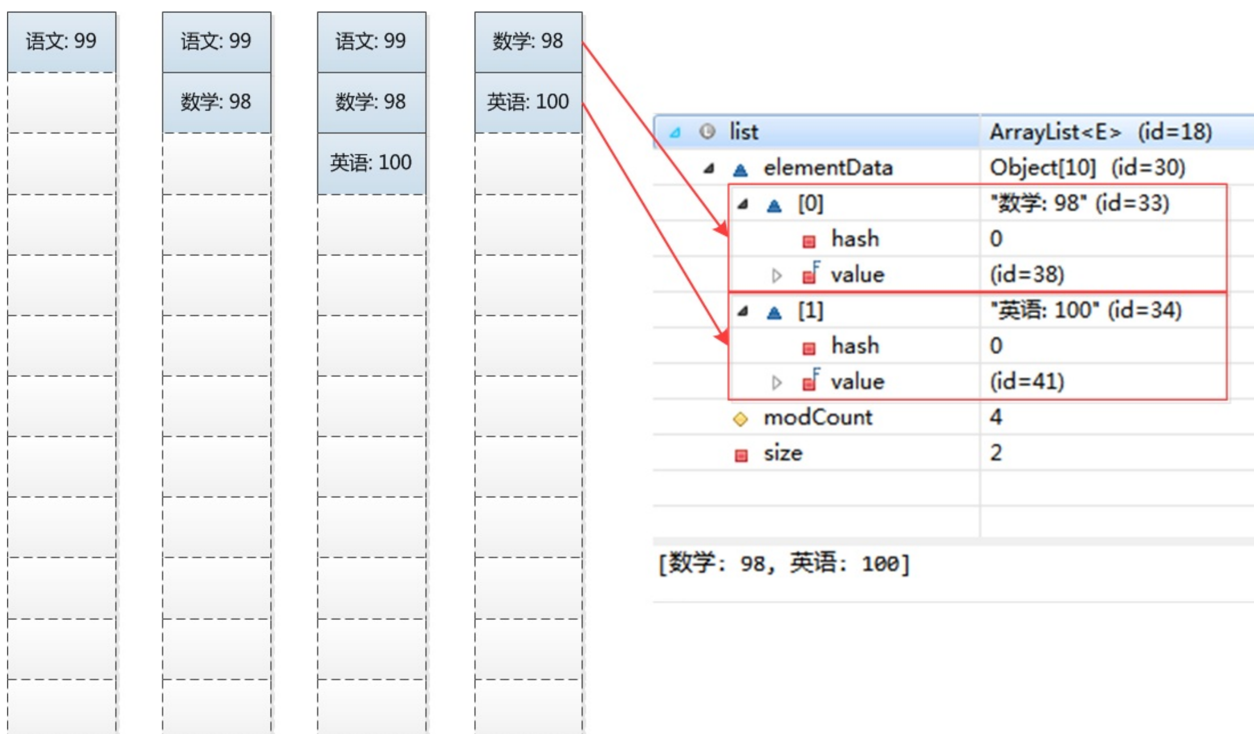
然后再来学习一下官方文档：

> **Resizable-array**implementation of the List interface. Implements all optional list operations, and permits all elements, including null. In addition to implementing the List interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to Vector, except that it is unsynchronized.)

ArrayList是一个相对来说比较简单的数据结构，最重要的一点就是它的自动扩容，可以认为就是我们常说的“动态数组”。
来看一段简单的代码：

```
ArrayList<String> list = new ArrayList<String>();
list.add("语文: 99");
list.add("数学: 98");
list.add("英语: 100");
list.remove(0);
```

在执行这四条语句时，是这么变化的：



其中，`add` 操作可以理解为直接将数组的内容置位，`remove` 操作可以理解为删除index为0的节点，并将后面元素移到0处。

二、add函数

当我们在ArrayList中增加元素的时候，会使用 `add` 函数。他会将元素放到末尾。具体实现如下：

```
public boolean add(E e) {
    ensureCapacityInternal(size + 1);  // Increments modCount!!
    elementData[size++] = e;
    return true;
}
```

我们可以看到他的实现其实最核心的内容就是 `ensureCapacityInternal` 。这个函数其实就是自动扩容机制的核心。我们依次来看一下他的具体实现

```java
private void ensureCapacityInternal(int minCapacity) {
    if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
        minCapacity = Math.max(DEFAULT_CAPACITY, minCapacity);
    }
    ensureExplicitCapacity(minCapacity);
}
private void ensureExplicitCapacity(int minCapacity) {
    modCount++;
    // overflow-conscious code
    if (minCapacity - elementData.length > 0)
        grow(minCapacity);
}
private void grow(int minCapacity) {
    // overflow-conscious code
    int oldCapacity = elementData.length;
    // 扩展为原来的1.5倍
    int newCapacity = oldCapacity + (oldCapacity >> 1);
    // 如果扩为1.5倍还不满足需求，直接扩为需求值
    if (newCapacity - minCapacity < 0)
        newCapacity = minCapacity;
    if (newCapacity - MAX_ARRAY_SIZE > 0)
        newCapacity = hugeCapacity(minCapacity);
    // minCapacity is usually close to size, so this is a win:
    elementData = Arrays.copyOf(elementData, newCapacity);
}
```

也就是说，当增加数据的时候，如果ArrayList的大小已经不满足需求时，那么就将数组变为原长度的1.5倍，之后的操作就是把老的数组拷到新的数组里面。例如，默认的数组大小是10，也就是说当我们 `add` 10个元素之后，再进行一次add时，就

会发生自动扩容，数组长度由10变为了15具体情况如下所示：

list	ArrayList<E> (id=18)
elementData	Object[10] (id=29)
[0]	"语文: 1" (id=37)
[1]	"数学: 2" (id=38)
[2]	"英语: 3" (id=39)
[3]	"政治: 4" (id=44)
[4]	"历史: 5" (id=45)
[5]	"物理: 6" (id=46)
[6]	"化学: 7" (id=47)
[7]	"生物: 8" (id=48)
[8]	"地理: 9" (id=49)
[9]	"体育: 10" (id=52)
modCount	10
size	10

list	ArrayList<E> (id=18)
elementData	Object[15] (id=55)
[0]	"语文: 1" (id=37)
[1]	"数学: 2" (id=38)
[2]	"英语: 3" (id=39)
[3]	"政治: 4" (id=44)
[4]	"历史: 5" (id=45)
[5]	"物理: 6" (id=46)
[6]	"化学: 7" (id=47)
[7]	"生物: 8" (id=48)
[8]	"地理: 9" (id=49)
[9]	"体育: 10" (id=52)
[10]	"信息: 11" (id=58)
modCount	11
size	11

三、set和get函数

Array的set和get函数就比较简单了，先做index检查，然后执行赋值或访问操作：

```
public E set(int index, E element) {
    rangeCheck(index);
    E oldValue = elementData(index);
    elementData[index] = element;
    return oldValue;
}
public E get(int index) {
    rangeCheck(index);
    return elementData(index);
}
```

四、remove函数

```
public E remove(int index) {
    rangeCheck(index);
    modCount++;
    E oldValue = elementData(index);
    int numMoved = size - index - 1;
    if (numMoved > 0)
        // 把后面的往前移
        System.arraycopy(elementData, index+1, elementData, inde
x,
                         numMoved);
    // 把最后的置null
    elementData[--size] = null; // clear to let GC do its work
    return oldValue;
}
```

一、概述

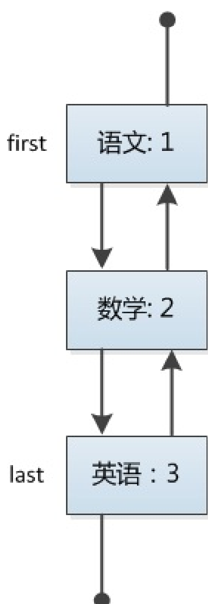
> 以双向链表实现。链表无容量限制，但双向链表本身使用了更多空间，也需要额外的链表指针操作。
>
> 按下标访问元素—get(i)/set(i,e) 要悲剧的遍历链表将指针移动到位(如果i>数组大小的一半，会从末尾移起)。
>
> 插入、删除元素时修改前后节点的指针即可，但还是要遍历部分链表的指针才能移动到下标所指的位置，只有在链表两头的操作—add()，addFirst()，removeLast()或用iterator()上的remove()能省掉指针的移动。

LinkedList是一个简单的数据结构，与ArrayList不同的是，他是基于链表实现的。

> Doubly-linked list implementation of the List and Deque interfaces. Implements all optional list operations, and permits all elements (including null).

```
LinkedList<String> list = new LinkedList<String>();
list.add("语文: 1");
list.add("数学: 2");
list.add("英语: 3");
```

结构也相对简单一些，如下图所示：



list	LinkedList<E> (id=17)
first	LinkedList$Node<E> (id=31)
item	"语文: 1" (id=35)
next	LinkedList$Node<E> (id=38)
prev	null
last	LinkedList$Node<E> (id=33)
item	"英语: 3" (id=41)
next	null
prev	LinkedList$Node<E> (id=38)
modCount	3
size	3

[语文: 1, 数学: 2, 英语: 3]

二、 set和get函数

```
public E set(int index, E element) {
    checkElementIndex(index);
    Node<E> x = node(index);
    E oldVal = x.item;
    x.item = element;
    return oldVal;
}
public E get(int index) {
    checkElementIndex(index);
    return node(index).item;
}
```

这两个函数都调用了 `node` 函数，该函数会以O(n/2)的性能去获取一个节点，具体实现如下所示：

```
Node<E> node(int index) {
    // assert isElementIndex(index);
    if (index < (size >> 1)) {
        Node<E> x = first;
        for (int i = 0; i < index; i++)
            x = x.next;
        return x;
    } else {
        Node<E> x = last;
        for (int i = size - 1; i > index; i--)
            x = x.prev;
        return x;
    }
}
```

就是判断index是在前半区间还是后半区间，如果在前半区间就从head搜索，而在后半区间就从tail搜索。而不是一直从头到尾的搜索。如此设计，将节点访问的复杂度由O(n)变为O(n/2)。

一、概述

从本文你可以学习到：

> 1. 什么时候会使用HashMap？他有什么特点？
> 2. 你知道HashMap的工作原理吗？
> 3. 你知道get和put的原理吗？equals()和hashCode()的都有什么作用？
> 4. 你知道hash的实现吗？为什么要这样实现？
> 5. 如果HashMap的大小超过了负载因子(load factor)定义的容量，怎么办？

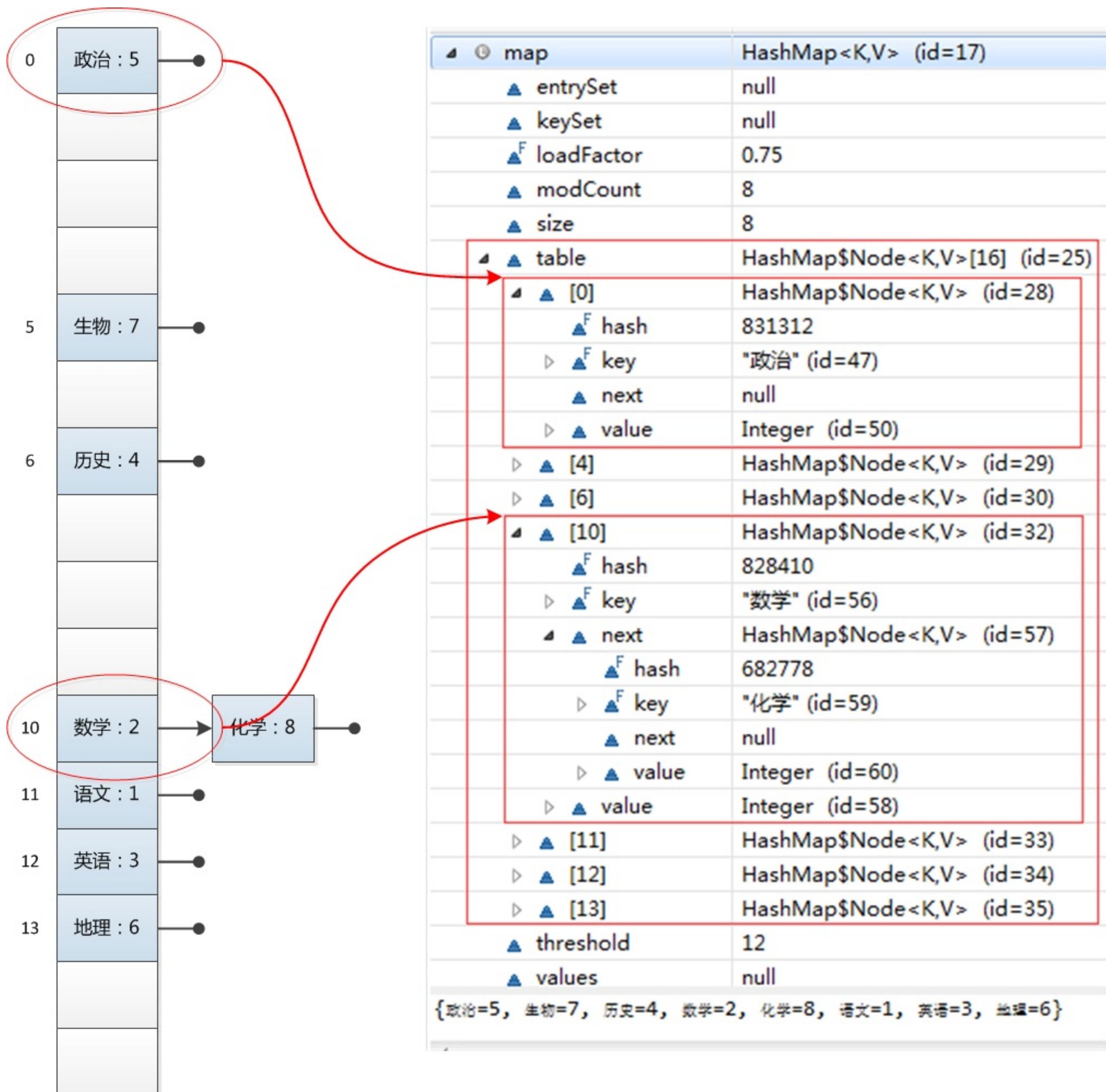
当我们执行下面的操作时：

```
HashMap<String, Integer> map = new HashMap<String, Integer>();
map.put("语文", 1);
map.put("数学", 2);
map.put("英语", 3);
map.put("历史", 4);
map.put("政治", 5);
map.put("地理", 6);
map.put("生物", 7);
map.put("化学", 8);
for(Entry<String, Integer> entry : map.entrySet()) {
    System.out.println(entry.getKey() + ": " + entry.getValue())
;
}
```

运行结果是

> 政治: 5
> 生物: 7
> 历史: 4
> 数学: 2
> 化学: 8
> 语文: 1
> 英语: 3
> 地理: 6

发生了什么呢？下面是一个大致的结构，希望我们对HashMap的结构有一个感性的认识：



在官方文档中是这样描述HashMap的：

> Hash table based**implementation of the Map interface**. This implementation provides all of the optional map operations, and permits null values and the null key. (The HashMap class is roughly equivalent to Hashtable, except that it is**unsynchronized**and**permits nulls**.) This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time.

几个关键的信息：基于Map接口实现、允许null键/值、非同步、不保证有序(比如插入的顺序)、也不保证序不随时间变化。

二、两个重要的参数

在HashMap中有两个很重要的参数，容量(Capacity)和负载因子(Load factor)

> - **Initial capacity** The capacity is **the number of buckets** in the hash table, The initial capacity is simply the capacity at the time the hash table is created.
> - **Load factor** The load factor is **a measure of how full the hash table is allowed to get** before its capacity is automatically increased.

简单的说，Capacity就是bucket的大小，Load factor就是bucket填满程度的最大比例。如果对迭代性能要求很高的话，不要把 `capacity` 设置过大，也不要把 `load factor` 设置过小。当bucket中的entries的数目大于 `capacity*load factor` 时就需要调整bucket的大小为当前的2倍。

三、put函数的实现

put函数大致的思路为：

1. 对key的hashCode()做hash，然后再计算index;
2. 如果没碰撞直接放到bucket里；
3. 如果碰撞了，以链表的形式存在buckets后；
4. 如果碰撞导致链表过长(大于等于 `TREEIFY_THRESHOLD`)，就把链表转换成红黑树；
5. 如果节点已经存在就替换old value(保证key的唯一性)
6. 如果bucket满了(超过 `load factor*current capacity`)，就要resize。

具体代码的实现如下：

```
public V put(K key, V value) {
    // 对key的hashCode()做hash
    return putVal(hash(key), key, value, false, true);
}
final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
               boolean evict) {
    Node<K,V>[] tab; Node<K,V> p; int n, i;
    // tab为空则创建
    if ((tab = table) == null || (n = tab.length) == 0)
```

```
        n = (tab = resize()).length;
    // 计算index，并对null做处理
    if ((p = tab[i = (n - 1) & hash]) == null)
        tab[i] = newNode(hash, key, value, null);
    else {
        Node<K,V> e; K k;
        // 节点存在
        if (p.hash == hash &&
            ((k = p.key) == key || (key != null && key.equals(k)
)))
            e = p;
        // 该链为树
        else if (p instanceof TreeNode)
            e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, k
ey, value);
        // 该链为链表
        else {
            for (int binCount = 0; ; ++binCount) {
                if ((e = p.next) == null) {
                    p.next = newNode(hash, key, value, null);
                    if (binCount >= TREEIFY_THRESHOLD - 1) // -1
 for 1st
                        treeifyBin(tab, hash);
                    break;
                }
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null && key.e
quals(k))))
                    break;
                p = e;
            }
        }
        // 写入
        if (e != null) { // existing mapping for key
            V oldValue = e.value;
            if (!onlyIfAbsent || oldValue == null)
                e.value = value;
            afterNodeAccess(e);
            return oldValue;
        }
```

```
        }
        ++modCount;
        // 超过load factor*current capacity，resize
        if (++size > threshold)
            resize();
        afterNodeInsertion(evict);
        return null;
    }
```

四、get函数的实现

在理解了put之后，get就很简单了。大致思路如下：

1. bucket里的第一个节点，直接命中；
2. 如果有冲突，则通过key.equals(k)去查找对应的entry

   若为树，则在树中通过key.equals(k)查找，O(logn)；

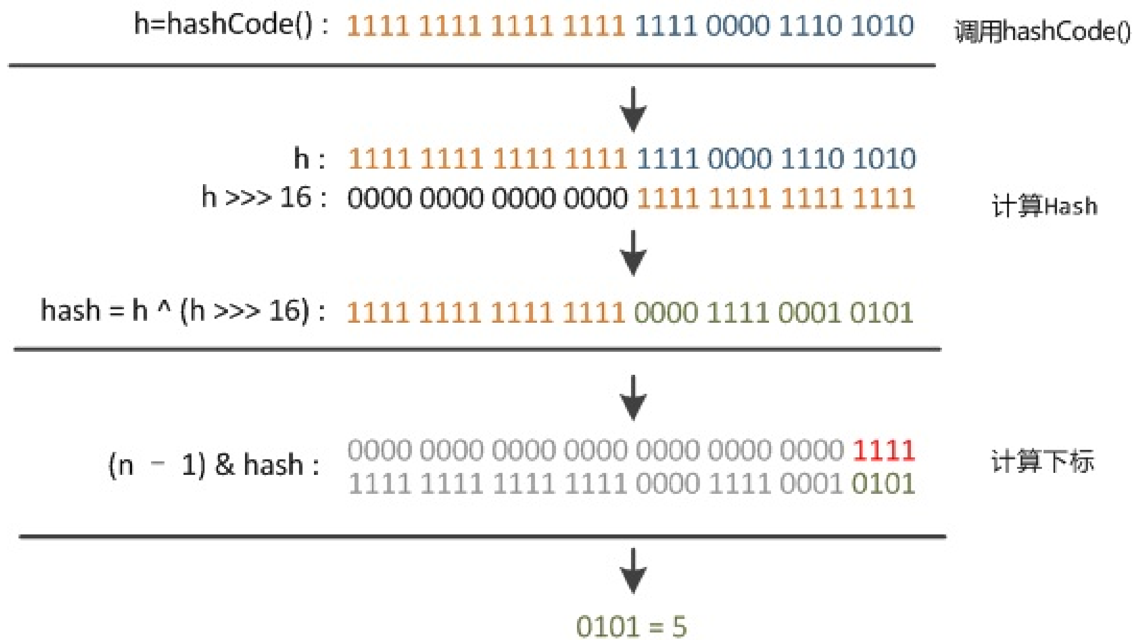
   若为链表，则在链表中通过key.equals(k)查找，O(n)。

具体代码的实现如下：

```
public V get(Object key) {
    Node<K,V> e;
    return (e = getNode(hash(key), key)) == null ? null : e.valu
e;
}
final Node<K,V> getNode(int hash, Object key) {
    Node<K,V>[] tab; Node<K,V> first, e; int n; K k;
    if ((tab = table) != null && (n = tab.length) > 0 &&
        (first = tab[(n - 1) & hash]) != null) {
        // 直接命中
        if (first.hash == hash && // always check first node
            ((k = first.key) == key || (key != null && key.equal
s(k))))
            return first;
        // 未命中
        if ((e = first.next) != null) {
            // 在树中get
            if (first instanceof TreeNode)
                return ((TreeNode<K,V>)first).getTreeNode(hash,
key);
            // 在链表中get
            do {
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null && key.e
quals(k))))
                    return e;
            } while ((e = e.next) != null);
        }
    }
    return null;
}
```

五、hash函数的实现

在get和put的过程中，计算下标时，先对hashCode进行hash操作，然后再通过hash值进一步计算下标，如下图所示：



在对hashCode()计算hash时具体实现是这样的：

```
static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16)
;
}
```

可以看到这个函数大概的作用就是：高16bit不变，低16bit和高16bit做了一个异或。其中代码注释是这样写的：

> Computes key.hashCode() and spreads (XORs) higher bits of hash to lower. Because the table uses power-of-two masking, sets of hashes that vary only in bits above the current mask will always collide. (Among known examples are sets of Float keys holding consecutive whole numbers in small tables.) So we apply a transform that spreads the impact of higher bits downward. There is a tradeoff between**speed, utility, and quality**of bit-spreading. Because many common sets of hashes are already**reasonably distributed**(so don't benefit from spreading), and because**we use trees to handle large sets of collisions in bins**, we just XOR some shifted bits in the cheapest possible way to reduce systematic lossage, as well as to incorporate impact of the highest bits that would otherwise never be used in index calculations because of table bounds.

在设计hash函数时，因为目前的table长度n为2的幂，而计算下标的时候，是这样实现的(使用 `&` 位操作，而非 `%` 求余)：

```
(n - 1) & hash
```

设计者认为这方法很容易发生碰撞。为什么这么说呢？不妨思考一下，在n - 1为15(0x1111)时，其实散列真正生效的只是低4bit的有效位，当然容易碰撞了。

因此，设计者想了一个顾全大局的方法(综合考虑了速度、作用、质量)，就是把高16bit和低16bit异或了一下。设计者还解释到因为现在大多数的hashCode的分布已经很不错了，就算是发生了碰撞也用 `O(logn)` 的tree去做了。仅仅异或一下，既减少了系统的开销，也不会造成的因为高位没有参与下标的计算(table长度比较小时)，从而引起的碰撞。

如果还是产生了频繁的碰撞，会发生什么问题呢？作者注释说，他们使用树来处理频繁的碰撞(we use trees to handle large sets of collisions in bins)，在JEP-180中，描述了这个问题：

> Improve the performance of java.util.HashMap under high hash-collision conditions by**using balanced trees rather than linked lists to store map entries**. Implement the same improvement in the LinkedHashMap class.

之前已经提过，在获取HashMap的元素时，基本分两步：

1. 首先根据hashCode()做hash，然后确定bucket的index；

2. 如果bucket的节点的key不是我们需要的，则通过keys.equals()在链中找。

在Java 8之前的实现中是用链表解决冲突的，在产生碰撞的情况下，进行get时，两步的时间复杂度是O(1)+O(n)。因此，当碰撞很厉害的时候n很大，O(n)的速度显然是影响速度的。

因此在Java 8中，利用红黑树替换链表，这样复杂度就变成了O(1)+O(logn)了，这样在n很大的时候，能够比较理想的解决这个问题，在Java 8：HashMap的性能提升一文中有性能测试的结果。

六、RESIZE的实现

当put时，如果发现目前的bucket占用程度已经超过了Load Factor所希望的比例，那么就会发生resize。在resize的过程，简单的说就是把bucket扩充为2倍，之后重新计算index，把节点再放到新的bucket中。resize的注释是这样描述的：

> Initializes or doubles table size. If null, allocates in accord with initial capacity target held in field threshold. Otherwise, because we are using power-of-two expansion, the elements from each bin must either**stay at same index**, or**move with a power of two offset**in the new table.

大致意思就是说，当超过限制的时候会resize，然而又因为我们使用的是2次幂的扩展(指长度扩为原来2倍)，所以，元素的位置要么是在原位置，要么是在原位置再移动2次幂的位置。

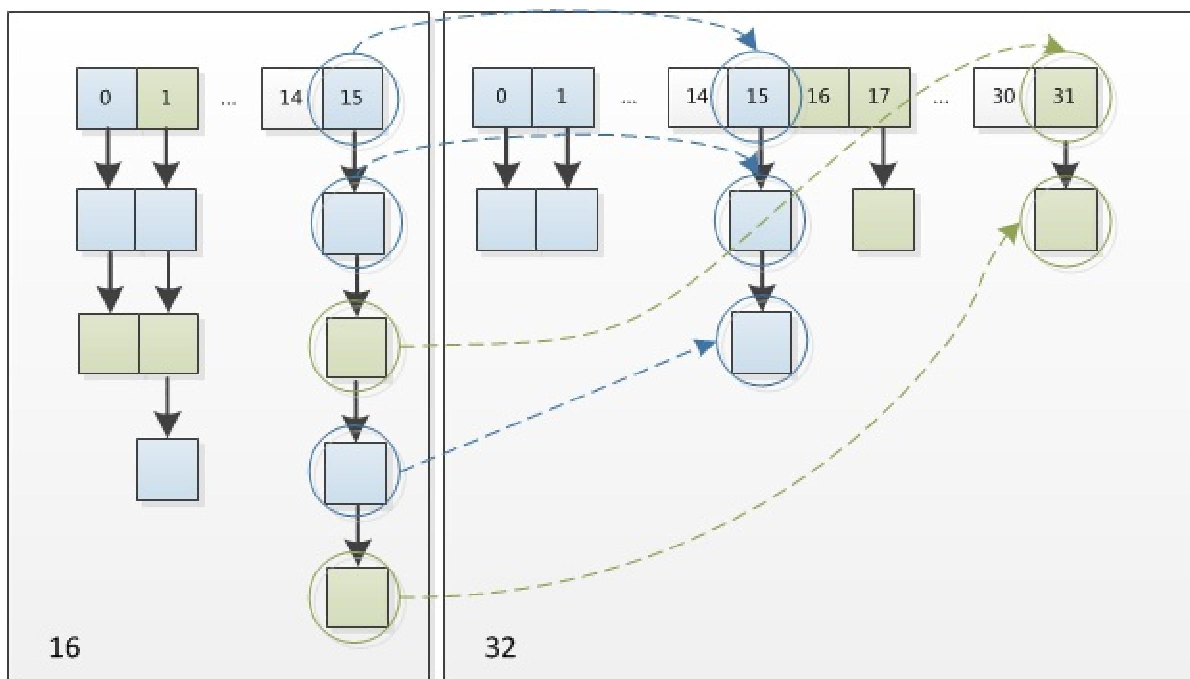
怎么理解呢？例如我们从16扩展为32时，具体的变化如下所示：

n - 1 0000 0000 0000 0000 0000 0000 0000 1111 → 1111 1111 1111 1111 0000 1111 0001 1111
hash1 1111 1111 1111 1111 0000 1111 0000 0101 → 1111 1111 1111 1111 0000 1111 0000 0101
hash2 1111 1111 1111 1111 0000 1111 0001 0101 → 1111 1111 1111 1111 0000 1111 0001 0101

因此元素在重新计算hash之后，因为n变为2倍，那么n-1的mask范围在高位多1bit(红色)，因此新的index就会发生这样的变化：

0101 = 5 —resize 16 => 2*16→
00101 = 5 原位置
10101 = 21 = 5 + 16 原位置 + oldCap

因此，我们在扩充HashMap的时候，不需要重新计算hash，只需要看看原来的hash值新增的那个bit是1还是0就好了，是0的话索引没变，是1的话索引变成“原索引+oldCap”。可以看看下图为16扩充为32的resize示意图：



这个设计确实非常的巧妙，既省去了重新计算hash值的时间，而且同时，由于新增的1bit是0还是1可以认为是随机的，因此resize的过程，均匀的把之前的冲突的节点分散到新的bucket了。

下面是代码的具体实现：

```
final Node<K,V>[] resize() {
    Node<K,V>[] oldTab = table;
    int oldCap = (oldTab == null) ? 0 : oldTab.length;
    int oldThr = threshold;
    int newCap, newThr = 0;
    if (oldCap > 0) {
        // 超过最大值就不再扩充了，就只好随你碰撞去吧
        if (oldCap >= MAXIMUM_CAPACITY) {
            threshold = Integer.MAX_VALUE;
            return oldTab;
        }
        // 没超过最大值，就扩充为原来的2倍
        else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
                 oldCap >= DEFAULT_INITIAL_CAPACITY)
            newThr = oldThr << 1; // double threshold
    }
    else if (oldThr > 0) // initial capacity was placed in thres
hold
        newCap = oldThr;
```

```
    else {              // zero initial threshold signifies usi
ng defaults
        newCap = DEFAULT_INITIAL_CAPACITY;
        newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAP
ACITY);
    }
    // 计算新的resize上限
    if (newThr == 0) {
        float ft = (float)newCap * loadFactor;
        newThr = (newCap < MAXIMUM_CAPACITY && ft < (float)MAXIM
UM_CAPACITY ?
                  (int)ft : Integer.MAX_VALUE);
    }
    threshold = newThr;
    @SuppressWarnings({"rawtypes","unchecked"})
        Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];
    table = newTab;
    if (oldTab != null) {
        // 把每个bucket都移动到新的buckets中
        for (int j = 0; j < oldCap; ++j) {
            Node<K,V> e;
            if ((e = oldTab[j]) != null) {
                oldTab[j] = null;
                if (e.next == null)
                    newTab[e.hash & (newCap - 1)] = e;
                else if (e instanceof TreeNode)
                    ((TreeNode<K,V>)e).split(this, newTab, j, ol
dCap);
                else { // preserve order
                    Node<K,V> loHead = null, loTail = null;
                    Node<K,V> hiHead = null, hiTail = null;
                    Node<K,V> next;
                    do {
                        next = e.next;
                        // 原索引
                        if ((e.hash & oldCap) == 0) {
                            if (loTail == null)
                                loHead = e;
                            else
                                loTail.next = e;
```

```
                        loTail = e;
                    }
                    // 原索引+oldCap
                    else {
                        if (hiTail == null)
                            hiHead = e;
                        else
                            hiTail.next = e;
                        hiTail = e;
                    }
                } while ((e = next) != null);
                // 原索引放到bucket里
                if (loTail != null) {
                    loTail.next = null;
                    newTab[j] = loHead;
                }
                // 原索引+oldCap放到bucket里
                if (hiTail != null) {
                    hiTail.next = null;
                    newTab[j + oldCap] = hiHead;
                }
            }
        }
    }
    }
    return newTab;
}
```

七、总结

我们现在可以回答开始的几个问题，加深对HashMap的理解：

1. 什么时候会使用**HashMap**？他有什么特点？
是基于Map接口的实现，存储键值对时，它可以接收null的键值，是非同步的，HashMap存储着Entry(hash, key, value, next)对象。

2. 你知道**HashMap**的工作原理吗？
通过hash的方法，通过put和get存储和获取对象。存储对象时，我们将K/V传给put方法时，它调用hashCode计算hash从而得到bucket位置，进一步存储，HashMap

会根据当前bucket的占用情况自动调整容量(超过 `Load Facotr` 则resize为原来的2倍)。获取对象时，我们将K传给get，它调用hashCode计算hash从而得到bucket位置，并进一步调用equals()方法确定键值对。如果发生碰撞的时候，Hashmap通过链表将产生碰撞冲突的元素组织起来，在Java 8中，如果一个bucket中碰撞冲突的元素超过某个限制(默认是8)，则使用红黑树来替换链表，从而提高速度。

3. 你知道**get**和**put**的原理吗？**equals()**和**hashCode()**的都有什么作用？
通过对key的hashCode()进行hashing，并计算下标(`(n-1) & hash`)，从而获得buckets的位置。如果产生碰撞，则利用key.equals()方法去链表或树中去查找对应的节点

4. 你知道**hash**的实现吗？为什么要这样实现？
在Java 1.8的实现中，是通过hashCode()的高16位异或低16位实现的： `(h = k.hashCode()) ^ (h >>> 16)` ，主要是从速度、功效、质量来考虑的，这么做可以在bucket的n比较小的时候，也能保证考虑到高低bit都参与到hash的计算中，同时不会有太大的开销。

5. 如果**HashMap**的大小超过了负载因子**(** `load factor` **)**定义的容量，怎么办？
如果超过了负载因子(默认**0.75**)，则会重新resize一个原来长度两倍的HashMap，并且重新调用hash方法。

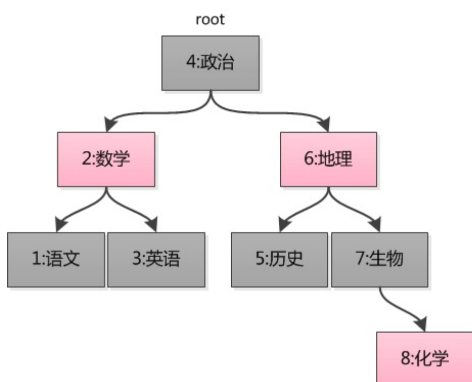
一、概述

> A **Red-Black tree** based NavigableMap implementation. The map is sorted according to the natural ordering of its keys, or by a Comparator provided at map creation time, depending on which constructor is used.
> This implementation provides guaranteed **log(n) time cost** for the containsKey, get, put and remove operations. Algorithms are adaptations of those in Cormen, Leiserson, and Rivest's Introduction to Algorithms.

之前已经学习过HashMap和LinkedHashMap了，HashMap不保证数据有序，LinkedHashMap保证数据可以保持插入顺序，而如果我们希望Map可以保持**key**的大小顺序的时候，我们就需要利用TreeMap了。

```
TreeMap<Integer, String> tmap = new TreeMap<Integer, String>();
tmap.put(1, "语文");
tmap.put(3, "英语");
tmap.put(2, "数学");
tmap.put(4, "政治");
tmap.put(5, "历史");
tmap.put(6, "地理");
tmap.put(7, "生物");
tmap.put(8, "化学");
for(Entry<Integer, String> entry : tmap.entrySet()) {
    System.out.println(entry.getKey() + ": " + entry.getValue())
;
}
```

其大致的结构如下所示：



Name	Value
tmap	TreeMap<K,V> (id=27)
comparator	null
descendingMap	null
entrySet	null
keySet	null
modCount	8
navigableKeySet	null
root	TreeMap$Entry<K,V> (id=31)
color	true
key	Integer (id=34)
left	TreeMap$Entry<K,V> (id=36)
parent	null
right	TreeMap$Entry<K,V> (id=39)
value	"政治" (id=40)
size	8
values	null

{1=语文, 2=数学, 3=英语, 4=政治, 5=历史, 6=地理, 7=生物, 8=化学}

使用红黑树的好处是能够使得树具有不错的平衡性，这样操作的速度就可以达到log(n)的水平了。具体红黑树的实现不在这里赘述，可以参考数据结构之红黑树、wikipedia-红黑树等的实现。

二、 put函数

> Associates the specified value with the specified key in this map.If the map previously contained a mapping for the key, the old value is replaced.

如果存在的话，old value被替换；如果不存在的话，则新添一个节点，然后对做红黑树的平衡操作。

```
public V put(K key, V value) {
    Entry<K,V> t = root;
    if (t == null) {
        compare(key, key); // type (and possibly null) check
        root = new Entry<>(key, value, null);
        size = 1;
        modCount++;
        return null;
    }
    int cmp;
    Entry<K,V> parent;
    // split comparator and comparable paths
    Comparator<? super K> cpr = comparator;
        // 如果该节点存在，则替换值直接返回
```

```
    if (cpr != null) {
        do {
            parent = t;
            cmp = cpr.compare(key, t.key);
            if (cmp < 0)
                t = t.left;
            else if (cmp > 0)
                t = t.right;
            else
                return t.setValue(value);
        } while (t != null);
    }
    else {
        if (key == null)
            throw new NullPointerException();
        @SuppressWarnings("unchecked")
            Comparable<? super K> k = (Comparable<? super K>) ke
y;
        do {
            parent = t;
            cmp = k.compareTo(t.key);
            if (cmp < 0)
                t = t.left;
            else if (cmp > 0)
                t = t.right;
            else
                return t.setValue(value);
        } while (t != null);
    }
    // 如果该节点未存在，则新建
    Entry<K,V> e = new Entry<>(key, value, parent);
    if (cmp < 0)
        parent.left = e;
    else
        parent.right = e;

    // 红黑树平衡调整
    fixAfterInsertion(e);
    size++;
    modCount++;
```

```
    return null;
}
```

三、 get函数

get函数则相对来说比较简单，以log(n)的复杂度进行get。

```
final Entry<K,V> getEntry(Object key) {
    // Offload comparator-based version for sake of performance
    if (comparator != null)
        return getEntryUsingComparator(key);
    if (key == null)
        throw new NullPointerException();
    @SuppressWarnings("unchecked")
        Comparable<? super K> k = (Comparable<? super K>) key;
    Entry<K,V> p = root;
        // 按照二叉树搜索的方式进行搜索，搜到返回
    while (p != null) {
        int cmp = k.compareTo(p.key);
        if (cmp < 0)
            p = p.left;
        else if (cmp > 0)
            p = p.right;
        else
            return p;
    }
    return null;
}
public V get(Object key) {
    Entry<K,V> p = getEntry(key);
    return (p==null ? null : p.value);
}
```

四、successor后继

TreeMap是如何保证其迭代输出是有序的呢？其实从宏观上来讲，就相当于树的中序遍历(LDR)。我们先看一下迭代输出的步骤

```
for(Entry<Integer, String> entry : tmap.entrySet()) {
    System.out.println(entry.getKey() + ": " + entry.getValue())
;
}
```

根据The enhanced for statement，for语句会做如下转换为：

```
for(Iterator<Map.Entry<String, String>> it = tmap.entrySet().ite
rator() ; tmap.hasNext(); ) {
    Entry<Integer, String> entry = it.next();
    System.out.println(entry.getKey() + ": " + entry.getValue())
;
}
```

在**it.next()**的调用中会使用**nextEntry**调用 `successor` 这个是过的后继的重点，具体实现如下：

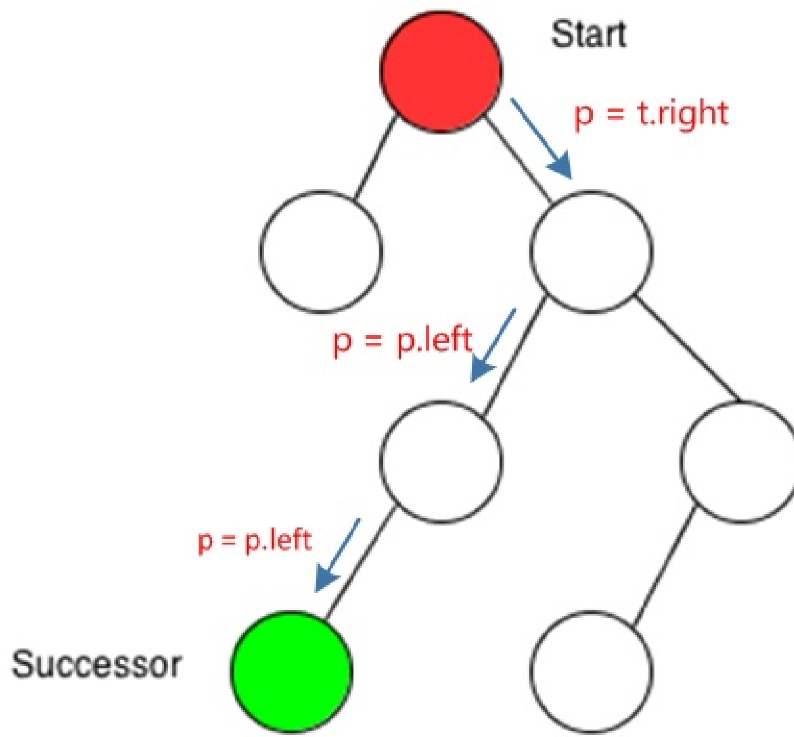
```
static <K,V> TreeMap.Entry<K,V> successor(Entry<K,V> t) {
    if (t == null)
        return null;
    else if (t.right != null) {
        // 有右子树的节点，后继节点就是右子树的“最左节点”
        // 因为“最左子树”是右子树的最小节点
        Entry<K,V> p = t.right;
        while (p.left != null)
            p = p.left;
        return p;
    } else {
        // 如果右子树为空，则寻找当前节点所在左子树的第一个祖先节点
        // 因为左子树找完了，根据LDR该D了
        Entry<K,V> p = t.parent;
        Entry<K,V> ch = t;
        // 保证左子树
        while (p != null && ch == p.right) {
            ch = p;
            p = p.parent;
        }
        return p;
    }
}
```

怎么理解这个successor呢？只要记住，这个是中序遍历就好了，L-D-R。具体细节如下：

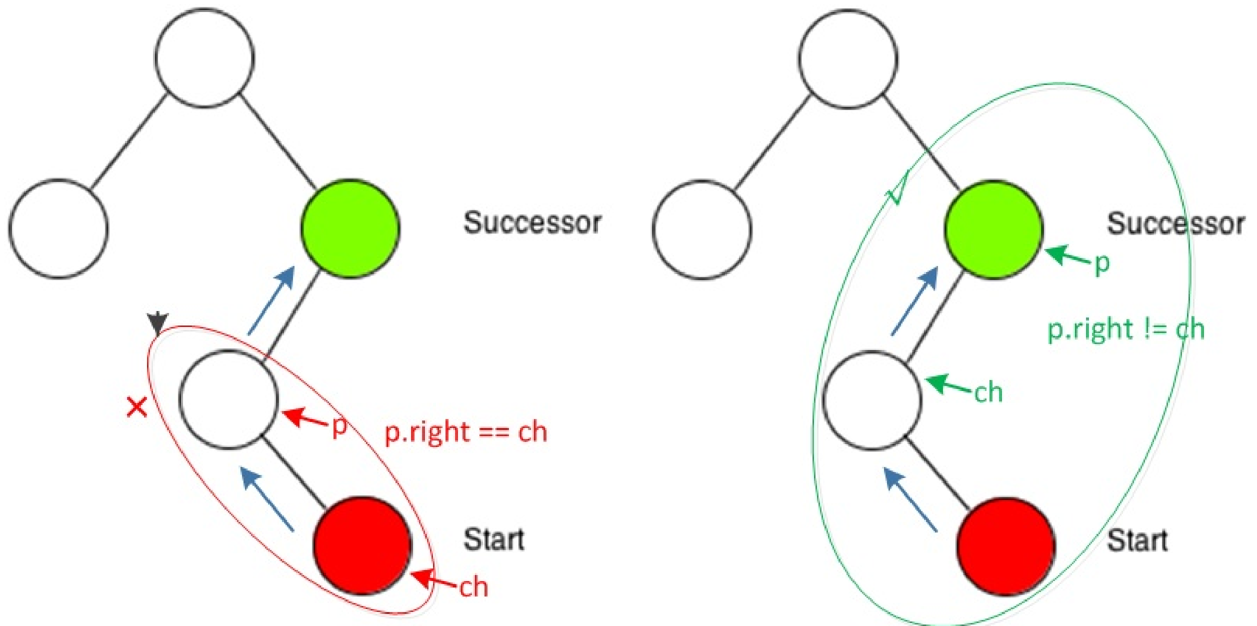
> a. 空节点，没有后继
> b. 有右子树的节点，后继就是右子树的“最左节点”
> c. 无右子树的节点，后继就是该节点所在左子树的第一个祖先节点

a.好理解，不过b, c，有点像绕口令啊，没关系，上图举个例子就懂了！

有右子树的节点，节点的下一个节点，肯定在右子树中，而右子树中“最左”的那个节点则是右子树中最小的一个，那么当然是右子树的“最左节点”，就好像下图所示：



无右子树的节点，先找到这个节点所在的左子树(右图)，那么这个节点所在的左子树的父节点(绿色节点)，就是下一个节点。



一、概述

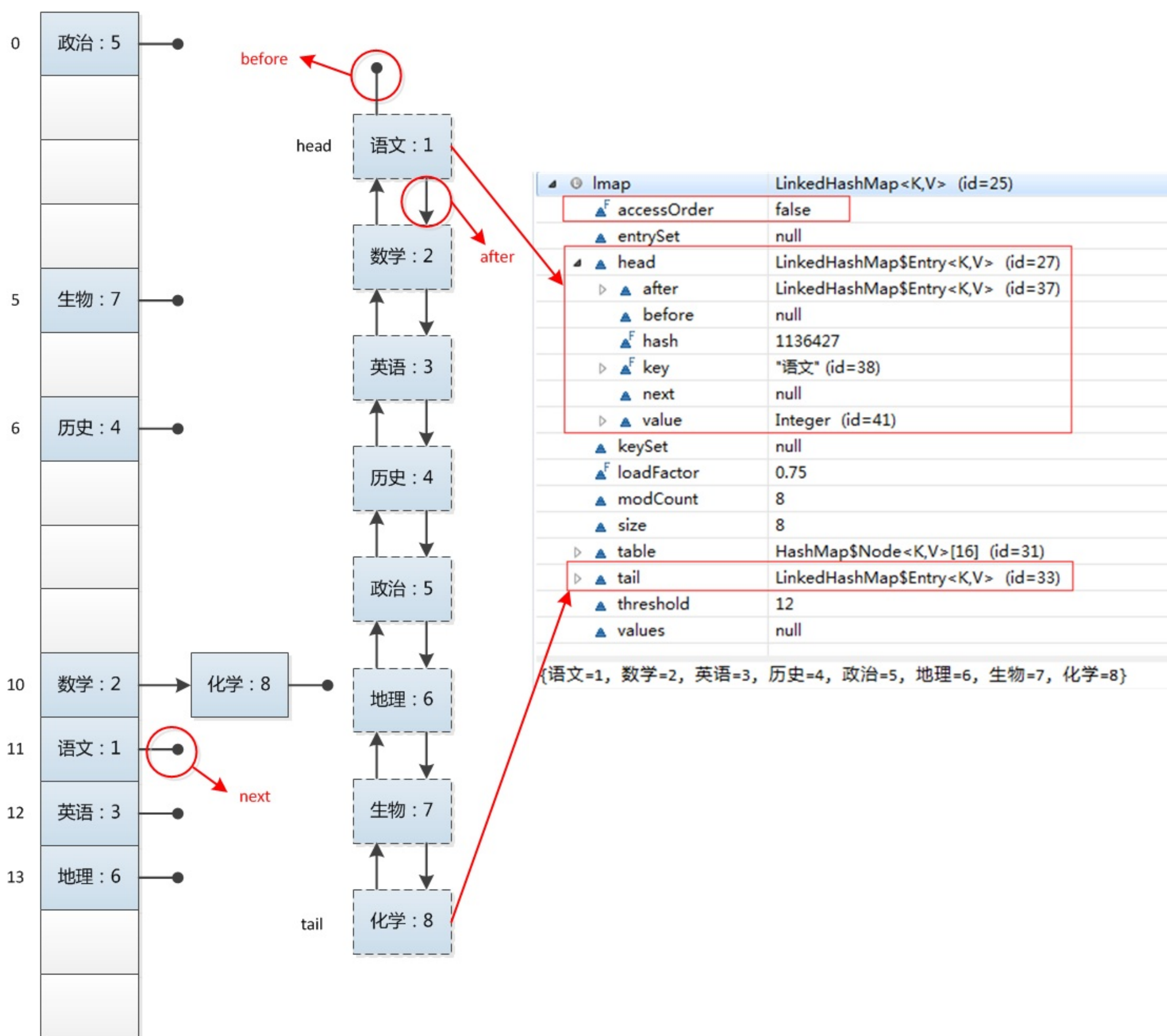
在理解了HashMap后，我们来学习LinkedHashMap的工作原理及实现。首先还是类似的，我们写一个简单的LinkedHashMap的程序：

```
LinkedHashMap<String, Integer> lmap = new LinkedHashMap<String,
Integer>();
lmap.put("语文", 1);
lmap.put("数学", 2);
lmap.put("英语", 3);
lmap.put("历史", 4);
lmap.put("政治", 5);
lmap.put("地理", 6);
lmap.put("生物", 7);
lmap.put("化学", 8);
for(Entry<String, Integer> entry : lmap.entrySet()) {
    System.out.println(entry.getKey() + ": " + entry.getValue())
;
}
```

运行结果是：

> 语文: 1 数学: 2 英语: 3 历史: 4 政治: 5 地理: 6 生物: 7 化学: 8

我们可以观察到，和HashMap的运行结果不同，LinkedHashMap的迭代输出的结果保持了插入顺序。是什么样的结构使得LinkedHashMap具有如此特性呢？我们还是一样的看看LinkedHashMap的内部结构，对它有一个感性的认识：



没错，正如官方文档所说：

> **Hash table** and **linked list** implementation of the Map interface, with predictable iteration order. This implementation differs from HashMap in that it maintains a **doubly-linked list** running through all of its entries. This linked list defines the iteration ordering, which is normally the order in which keys were inserted into the map (**insertion-order**).

LinkedHashMap是Hash表和链表的实现，并且依靠着双向链表保证了迭代顺序是插入的顺序。

二、 三个重点实现的函数

在HashMap中提到了下面的定义：

```
// Callbacks to allow LinkedHashMap post-actions
void afterNodeAccess(Node<K,V> p) { }
void afterNodeInsertion(boolean evict) { }
void afterNodeRemoval(Node<K,V> p) { }
```

LinkedHashMap继承于HashMap，因此也重新实现了这3个函数，顾名思义这三个函数的作用分别是：节点访问后、节点插入后、节点移除后做一些事情。

afterNodeAccess函数

```
void afterNodeAccess(Node<K,V> e) { // move node to last
    LinkedHashMap.Entry<K,V> last;
    // 如果定义了accessOrder，那么就保证最近访问节点放到最后
    if (accessOrder && (last = tail) != e) {
        LinkedHashMap.Entry<K,V> p =
            (LinkedHashMap.Entry<K,V>)e, b = p.before, a = p.aft
er;
        p.after = null;
        if (b == null)
            head = a;
        else
            b.after = a;
        if (a != null)
            a.before = b;
        else
            last = b;
        if (last == null)
            head = p;
        else {
            p.before = last;
            last.after = p;
        }
        tail = p;
        ++modCount;
    }
}
```

就是说在进行put之后就算是对节点的访问了，那么这个时候就会更新链表，把最近访问的放到最后，保证链表。

afterNodeInsertion函数

```
void afterNodeInsertion(boolean evict) { // possibly remove elde
st
    LinkedHashMap.Entry<K,V> first;
    // 如果定义了溢出规则，则执行相应的溢出
    if (evict && (first = head) != null && removeEldestEntry(fir
st)) {
        K key = first.key;
        removeNode(hash(key), key, null, false, true);
    }
}
```

如果用户定义了 `removeEldestEntry` 的规则，那么便可以执行相应的移除操作。

afterNodeRemoval函数

```
void afterNodeRemoval(Node<K,V> e) { // unlink
    // 从链表中移除节点
    LinkedHashMap.Entry<K,V> p =
        (LinkedHashMap.Entry<K,V>)e, b = p.before, a = p.after;
    p.before = p.after = null;
    if (b == null)
        head = a;
    else
        b.after = a;
    if (a == null)
        tail = b;
    else
        a.before = b;
}
```

这个函数是在移除节点后调用的，就是将节点从双向链表中删除。

我们从上面3个函数看出来，基本上都是为了保证双向链表中的节点次序或者双向链表容量所做的一些额外的事情，目的就是保持双向链表中节点的顺序要从eldest到youngest。

三、put和get函数

`put` 函数在LinkedHashMap中未重新实现，只是实现了 `afterNodeAccess` 和 `afterNodeInsertion` 两个回调函数。 `get` 函数则重新实现并加入了 `afterNodeAccess` 来保证访问顺序，下面是 `get` 函数的具体实现：

```
public V get(Object key) {
    Node<K,V> e;
    if ((e = getNode(hash(key), key)) == null)
        return null;
    if (accessOrder)
        afterNodeAccess(e);
    return e.value;
}
```

值得注意的是，在accessOrder模式下，只要执行get或者put等操作的时候，就会产生 `structural modification` 。官方文档是这么描述的：

> A structural modification is any operation that adds or deletes one or more mappings or, in the case of access-ordered linked hash maps, affects iteration order. In insertion-ordered linked hash maps, merely changing the value associated with a key that is already contained in the map is not a structural modification. **In access-ordered linked hash maps, merely querying the map with get is a structural modification**.

总之，LinkedHashMap不愧是HashMap的儿子，和老子太像了，当然，青出于蓝而胜于蓝，LinkedHashMap的其他的操作也基本上都是为了维护好那个具有访问顺序的双向链表。

一、泛型简介

1.引入泛型的目的

了解引入泛型的动机，就先从语法糖开始了解。

语法糖

语法糖（Syntactic Sugar），也称糖衣语法，是由英国计算机学家Peter.J.Landin发明的一个术语，指在计算机语言中添加的某种语法，这种语法对语言的功能并没有影响，但是更方便程序员使用。Java中最常用的语法糖主要有泛型、变长参数、条件编译、自动拆装箱、内部类等。虚拟机并不支持这些语法，它们在编译阶段就被还原回了简单的基础语法结构，这个过程成为解语法糖。

泛型的目的： Java 泛型就是把一种语法糖，通过泛型使得在编译阶段完成一些类型转换的工作，避免在运行时强制类型转换而出现 `ClassCastException` ，即类型转换异常。

2.泛型初探

JDK 1.5 时才增加了泛型，并在很大程度上都是方便集合的使用，使其能够记住其元素的数据类型。

在泛型（Generic type或Generics）出现之前，是这么写代码的：

```
public static void main(String[] args)
{
    List list = new ArrayList();
    list.add("123");
    list.add("456");

    System.out.println((String)list.get(0));
}
```

当然这是完全允许的，因为List里面的内容是Object类型的，自然任何对象类型都可以放入、都可以取出，但是这么写会有两个问题：

1、当一个对象放入集合时，集合不会记住此对象的类型，当再次从集合中取出此对象时，该对象的编译类型变成了Object。

2、运行时需要人为地强制转换类型到具体目标，实际的程序绝不会这么简单，一个不小心就会出现java.lang.ClassCastException。

所以，泛型出现之后，上面的代码就改成了大家都熟知的写法：

```
public static void main(String[] args)
{
    List<String>
    list = new ArrayList<String>();
    list.add("123");
    list.add("456");

    System.out.println(list.get(0));
}
```

这就是泛型。泛型是对Java语言类型系统的一种扩展，有点类似于C++的模板，可以把类型参数看作是使用参数化类型时指定的类型的一个占位符。引入泛型，是对Java语言一个较大的功能增强，带来了很多的好处。

3.泛型的好处

①类型安全。类型错误现在在编译期间就被捕获到了，而不是在运行时当作java.lang.ClassCastException展示出来，将类型检查从运行时挪到编译时有助于开发者更容易找到错误，并提高程序的可靠性。

②消除了代码中许多的强制类型转换，增强了代码的可读性。

③为较大的优化带来了可能。

二、泛型的使用

1.泛型类和泛型接口

下面是JDK 1.5 以后，List接口，以及ArrayList类的代码片段。

```
//定义接口时指定了一个类型形参，该形参名为E
public interface List<E> extends Collection<E> {
    //在该接口里，E可以作为类型使用
    public E get(int index) {}
    public void add(E e) {}
}

//定义类时指定了一个类型形参，该形参名为E
public class ArrayList<E> extends AbstractList<E> implements List
<E> {
    //在该类里，E可以作为类型使用
    public void set(E e) {
    .......................
    }
}
```

这就是泛型的实质：允许在定义接口、类时声明类型形参，类型形参在整个接口、类体内可当成类型使用，几乎所有可使用普通类型的地方都可以使用这种类型形参。

下面具体讲解泛型类的使用。泛型接口的使用与泛型类几乎相同，可以比对自行学习。

泛型类

定义一个容器类，存放键值对key-value，键值对的类型不确定，可以使用泛型来定义，分别指定为K和V。

```
public class Container<K, V> {

    private K key;
    private V value;

    public Container(K k, V v) {
        key = k;
        value = v;
    }

    public K getkey() {
        return key;
    }

    public V getValue() {
        return value;
    }

    public void setKey() {
        this.key = key;
    }

    public void setValue() {
        this.value = value;
    }

}
```

在使用Container类时，只需要指定K，V的具体类型即可，从而创建出逻辑上不同的Container实例，用来存放不同的数据类型。

```
 public static void main(String[] args) {
      Container<String,String>  c1=new Container<String ,String>(
"name","hello");
      Container<String,Integer> c2=new Container<String,Integer>(
"age",22);
      Container<Double,Double>  c3=new Container<Double,Double>(1
.1,1.3);
      System.out.println(c1.getKey() + " : " + c1.getValue());

      System.out.println(c2.getKey() + " : " + c2.getValue());

      System.out.println(c3.getKey() + " : " + c3.getValue());
 }
```

在JDK 1.7 增加了泛型的“菱形”语法：**Java**允许在构造器后不需要带完成的泛型信息，只要给出一对尖括号（**<>**）即可，**Java**可以推断尖括号里应该是什么泛型信息。
如下所示：

```
Container<String,String> c1=new Container<>("name","hello");
Container<String,Integer> c2=new Container<>("age",22);
```

泛型类派生子类

当创建了带泛型声明的接口、父类之后，可以为该接口创建实现类，或者从该父类派生子类，需要注意：使用这些接口、父类派生子类时不能再包含类型形参，需要传入具体的类型。
错误的方式：

```
public class A extends Container<K, V>{}
```

正确的方式：

```
public class A extends Container<Integer, String>{}
```

也可以不指定具体的类型，如下：

```
public class A extends Container{}
```

此时系统会把K,V形参当成Object类型处理。

2.泛型的方法

前面在介绍泛型类和泛型接口中提到，可以在泛型类、泛型接口的方法中，把泛型中声明的类型形参当成普通类型使用。 如下面的方式：

```
public class Container<K, V>
 {
.......................
    public K getkey() {
        return key;
    }
    public void setKey() {
        this.key = key;
    }
....................
}
```

但在另外一些情况下，在类、接口中没有使用泛型时，定义方法时想定义类型形参，就会使用泛型方法。如下方式：

```
public class Main{
  public static <T> void out(T t){
       System.out.println(t);
  }
  public static void main(String[] args){
       out("hansheng");
       out(123);
  }
}
```

所谓泛型方法，就是在声明方法时定义一个或多个类型形参。 泛型方法的用法格式如下：

```
修饰符<T, S> 返回值类型 方法名（形参列表）｛
    方法体
｝
```

注意： 方法声明中定义的形参只能在该方法里使用，而接口、类声明中定义的类型形参则可以在整个接口、类中使用。

```java
class Demo{
  public <T> T fun(T t){   // 可以接收任意类型的数据
   return t ;     // 直接把参数返回
  }
};
public class GenericsDemo26{
  public static void main(String args[]){
    Demo d = new Demo() ; // 实例化Demo对象
    String str = d.fun("汤姆") ; // 传递字符串
    int i = d.fun(30) ;  // 传递数字，自动装箱
    System.out.println(str) ; // 输出内容
    System.out.println(i) ;  // 输出内容
  }
};
```

当调用 `fun()` 方法时，根据传入的实际对象，编译器就会判断出类型形参T所代表的实际类型。

3.泛型构造器

正如泛型方法允许在方法签名中声明类型形参一样，Java也允许在构造器签名中声明类型形参，这样就产生了所谓的泛型构造器。
和使用普通泛型方法一样没区别，一种是显式指定泛型参数，另一种是隐式推断，如果是显式指定则以显式指定的类型参数为准，如果传入的参数的类型和指定的类型实参不符，将会编译报错。

```java
public class Person {
    public <T> Person(T t) {
        System.out.println(t);
    }

}
```

```java
public static void main(String[] args){
        //隐式
        new Person(22);
        //显示
        new<String> Person("hello");
}
```

这里唯一需要特殊注明的就是，如果构造器是泛型构造器，同时该类也是一个泛型类的情况下应该如何使用泛型构造器：
因为泛型构造器可以显式指定自己的类型参数（需要用到菱形，放在构造器之前），而泛型类自己的类型实参也需要指定（菱形放在构造器之后），这就同时出现了两个菱形了，这就会有一些小问题，具体用法再这里总结一下。
以下面这个例子为代表

```java
public class Person<E> {
    public <T> Person(T t) {
        System.out.println(t);
    }

}
```

这种用法：`Person<String> a = new <Integer>Person<>(15);` 这种语法不允许，会直接编译报错！

三、类型通配符

顾名思义就是匹配任意类型的类型实参。

类型通配符是一个问号（？)，将一个问号作为类型实参传给List集合，写作： `List<?>` （意思是元素类型未知的List）。这个问号（？）被成为通配符，它的元素类型可以匹配任何类型。

```
public void test(List<?> c){
  for(int i =0;i<c.size();i++){
    System.out.println(c.get(i));
  }
}
```

现在可以传入任何类型的List来调用test()方法，程序依然可以访问集合c中的元素，其类型是Object。

```
List<?> c = new ArrayList<String>();
//编译器报错
c.add(new Object());
```

但是并不能把元素加入到其中。因为程序无法确定c集合中元素的类型，所以不能向其添加对象。
下面就该引入带限通配符，来确定集合元素中的类型。

带限通配符

简单来讲，使用通配符的目的是来限制泛型的类型参数的类型，使其满足某种条件，固定为某些类。

主要分为两类即：上限通配符和下限通配符。

1.上限通配符

如果想限制使用泛型类别时，只能用某个特定类型或者是其子类型才能实例化该类型时，可以在定义类型时，使用**extends**关键字指定这个类型必须是继承某个类，或者实现某个接口，也可以是这个类或接口本身。

```
它表示集合中的所有元素都是Shape类型或者其子类
List<? extends Shape>
```

这就是所谓的上限通配符，使用关键字extends来实现，实例化时，指定类型实参只能是extends后类型的子类或其本身。
例如：

```
//Circle是其子类
List<? extends Shape> list = new ArrayList<Circle>();
```

这样就确定集合中元素的类型，虽然不确定具体的类型，但最起码知道其父类。然后进行其他操作。

2.下限通配符

如果想限制使用泛型类别时，只能用某个特定类型或者是其父类型才能实例化该类型时，可以在定义类型时，使用**super**关键字指定这个类型必须是是某个类的父类，或者是某个接口的父接口，也可以是这个类或接口本身。

```
它表示集合中的所有元素都是Circle类型或者其父类
List <? super Circle>
```

这就是所谓的下限通配符，使用关键字super来实现，实例化时，指定类型实参只能是extends后类型的子类或其本身。
例如：

```
//Shape是其父类
List<? super Circle> list = new ArrayList<Shape>();
```

四、类型擦除

```
Class c1=new ArrayList<Integer>().getClass();
Class c2=new ArrayList<String>().getClass();
System.out.println(c1==c2);
```

程序输出：

> true。

这是因为不管为泛型的类型形参传入哪一种类型实参，对于Java来说，它们依然被当成同一类处理，在内存中也只占用一块内存空间。从Java泛型这一概念提出的目的来看，其只是作用于代码编译阶段，在编译过程中，对于正确检验泛型结果后，会将泛型的相关信息擦出，也就是说，成功编译过后的class文件中是不包含任何泛型信息的。泛型信息不会进入到运行时阶段。

在静态方法、静态初始化块或者静态变量的声明和初始化中不允许使用类型形参。由于系统中并不会真正生成泛型类，所以**instanceof**运算符后不能使用泛型类。

一、概述

Java反射机制定义

Java反射机制是在运行状态中，对于任意一个类，都能够知道这个类中的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能称为java语言的反射机制。
Java 反射机制的功能

1.在运行时判断任意一个对象所属的类。

2.在运行时构造任意一个类的对象。

3.在运行时判断任意一个类所具有的成员变量和方法。

4.在运行时调用任意一个对象的方法。

5.生成动态代理。

Java 反射机制的应用场景

1.逆向代码 ，例如反编译

2.与注解相结合的框架 例如Retrofit

3.单纯的反射机制应用框架 例如EventBus

4.动态生成类框架 例如Gson

二、通过**Java**反射查看类信息

获得**Class**对象
每个类被加载之后，系统就会为该类生成一个对应的Class对象。通过该Class对象就可以访问到JVM中的这个类。

在Java程序中获得Class对象通常有如下三种方式：

1.使用Class类的forName(String clazzName)静态方法。该方法需要传入字符串参数，该字符串参数的值是某个类的全限定名（必须添加完整包名）。

2.调用某个类的class属性来获取该类对应的Class对象。

3.调用某个对象的getClass()方法。该方法是java.lang.Object类中的一个方法。

```
//第一种方式 通过Class类的静态方法——forName()来实现
class1 = Class.forName("com.lvr.reflection.Person");
//第二种方式 通过类的class属性
class1 = Person.class;
//第三种方式 通过对象getClass方法
Person person = new Person();
Class<?> class1 = person.getClass();
```

获取**class**对象的属性、方法、构造函数等

1.获取class对象的成员变量

```
Field[] allFields = class1.getDeclaredFields();//获取class对象的所有属性
Field[] publicFields = class1.getFields();//获取class对象的public属性
Field ageField = class1.getDeclaredField("age");//获取class指定属性

Field desField = class1.getField("des");//获取class指定的public属性
```

2.获取class对象的方法

```
Method[] methods = class1.getDeclaredMethods();//获取class对象的所有声明方法
Method[] allMethods = class1.getMethods();//获取class对象的所有public方法 包括父类的方法
Method method = class1.getMethod("info", String.class);//返回次Class对象对应类的、带指定形参列表的public方法
Method declaredMethod = class1.getDeclaredMethod("info", String.class);//返回次Class对象对应类的、带指定形参列表的方法
```

3.获取class对象的构造函数

```
Constructor<?>[] allConstructors = class1.getDeclaredConstructors();//获取class对象的所有声明构造函数
Constructor<?>[] publicConstructors = class1.getConstructors();//获取class对象public构造函数
Constructor<?> constructor = class1.getDeclaredConstructor(String.class);//获取指定声明构造函数
Constructor publicConstructor = class1.getConstructor(String.class);//获取指定声明的public构造函数
```

4.其他方法

```
Annotation[] annotations = (Annotation[]) class1.getAnnotations();//获取class对象的所有注解
Annotation annotation = (Annotation) class1.getAnnotation(Deprecated.class);//获取class对象指定注解
Type genericSuperclass = class1.getGenericSuperclass();//获取class对象的直接超类的 Type
Type[] interfaceTypes = class1.getGenericInterfaces();//获取class对象的所有接口的type集合
```

获取**class**对象的信息

比较多。

```
boolean isPrimitive = class1.isPrimitive();//判断是否是基础类型
boolean isArray = class1.isArray();//判断是否是集合类
boolean isAnnotation = class1.isAnnotation();//判断是否是注解类
boolean isInterface = class1.isInterface();//判断是否是接口类
boolean isEnum = class1.isEnum();//判断是否是枚举类
boolean isAnonymousClass = class1.isAnonymousClass();//判断是否是
匿名内部类
boolean isAnnotationPresent = class1.isAnnotationPresent(Depreca
ted.class);//判断是否被某个注解类修饰
String className = class1.getName();//获取class名字 包含包名路径
Package aPackage = class1.getPackage();//获取class的包信息
String simpleName = class1.getSimpleName();//获取class类名
int modifiers = class1.getModifiers();//获取class访问权限
Class<?>[] declaredClasses = class1.getDeclaredClasses();//内部类
Class<?> declaringClass = class1.getDeclaringClass();//外部类
```

三、通过Java反射生成并操作对象

生成类的实例对象

1.使用Class对象的newInstance()方法来创建该Class对象对应类的实例。这种方式要求该Class对象的对应类有默认构造器，而执行newInstance()方法时实际上是利用默认构造器来创建该类的实例。

2.先使用Class对象获取指定的Constructor对象，再调用Constructor对象的newInstance()方法来创建该Class对象对应类的实例。通过这种方式可以选择使用指定的构造器来创建实例。

```
//第一种方式 Class对象调用newInstance()方法生成
Object obj = class1.newInstance();
//第二种方式 对象获得对应的Constructor对象，再通过该Constructor对象的ne
wInstance()方法生成
Constructor<?> constructor = class1.getDeclaredConstructor(Strin
g.class);//获取指定声明构造函数
obj = constructor.newInstance("hello");
```

调用类的方法

1.通过Class对象的getMethods()方法或者getMethod()方法获得指定方法，返回Method数组或对象。

2.调用Method对象中的 `Object invoke(Object obj, Object... args)` 方法。第一个参数对应调用该方法的实例对象，第二个参数对应该方法的参数。

```
 // 生成新的对象：用newInstance()方法
 Object obj = class1.newInstance();
//首先需要获得与该方法对应的Method对象
Method method = class1.getDeclaredMethod("setAge", int.class);
//调用指定的函数并传递参数
method.invoke(obj, 28);
```

当通过**Method**的**invoke()**方法来调用对应的方法时，**Java**会要求程序必须有调用该方法的权限。如果程序确实需要调用某个对象的**private**方法，则可以先调用**Method**对象的如下方法。
setAccessible(boolean flag)：将**Method**对象的**acessible**设置为指定的布尔值。值为**true**，指示该**Method**在使用时应该取消**Java**语言的访问权限检查；值为**false**，则知识该**Method**在使用时要实施**Java**语言的访问权限检查。

访问成员变量值

1.通过Class对象的getFields()方法或者getField()方法获得指定方法，返回Field数组或对象。

2.Field提供了两组方法来读取或设置成员变量的值：
getXXX(Object obj):获取obj对象的该成员变量的值。此处的XXX对应8种基本类型。如果该成员变量的类型是引用类型，则取消get后面的XXX。
setXXX(Object obj,XXX val)：将obj对象的该成员变量设置成val值。

```
//生成新的对象：用newInstance()方法
Object obj = class1.newInstance();
//获取age成员变量
Field field = class1.getField("age");
//将obj对象的age的值设置为10
field.setInt(obj, 10);
//获取obj对象的age的值
field.getInt(obj);
```

一、代理模式

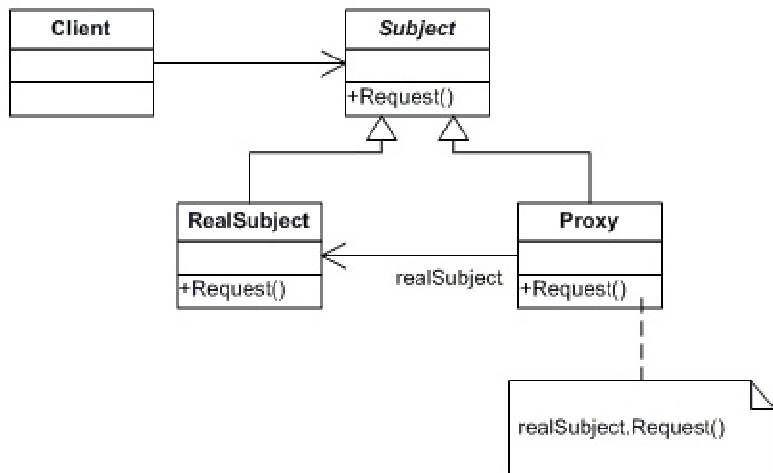
定义：给某个对象提供一个代理对象，并由代理对象控制对于原对象的访问，即客户不直接操控原对象，而是通过代理对象间接地操控原对象。

1、代理模式的理解

代理模式使用代理对象完成用户请求，屏蔽用户对真实对象的访问。现实世界的代理人被授权执行当事人的一些事宜，无需当事人出面，从第三方的角度看，似乎当事人并不存在，因为他只和代理人通信。而事实上代理人是要有当事人的授权，并且在核心问题上还需要请示当事人。
在软件设计中，使用代理模式的意图也很多，比如因为安全原因需要屏蔽客户端直接访问真实对象，或者在远程调用中需要使用代理类处理远程方法调用的技术细节，也可能为了提升系统性能，对真实对象进行封装，从而达到延迟加载的目的。

2、代理模式的参与者

代理模式的角色分四种：



主题接口：Subject 是委托对象和代理对象都共同实现的接口，即代理类的所实现的行为接口。Request() 是委托对象和代理对象共同拥有的方法。
目标对象：ReaSubject 是原对象，也就是被代理的对象。
代理对象：Proxy 是代理对象，用来封装真是主题类的代理类。
客户端 ：使用代理类和主题接口完成一些工作。

3、代理模式的分类

代理的实现分为：

静态代理：代理类是在编译时就实现好的。也就是说 Java 编译完成后代理类是一个实际的 class 文件。
动态代理：代理类是在运行时生成的。也就是说 Java 编译完之后并没有实际的 class 文件，而是在运行时动态生成的类字节码，并加载到JVM中。

4、代理模式的实现思路

1.代理对象和目标对象均实现同一个行为接口。

2.代理类和目标类分别具体实现接口逻辑。

3.在代理类的构造函数中实例化一个目标对象。

4.在代理类中调用目标对象的行为接口。

5.客户端想要调用目标对象的行为接口，只能通过代理类来操作。

5、静态代理模式的简单实现

```
public class ProxyDemo {
    public static void main(String args[]){
        RealSubject subject = new RealSubject();
        Proxy p = new Proxy(subject);
        p.request();
    }
}

interface Subject{
    void request();
}

class RealSubject implements Subject{
    public void request(){
        System.out.println("request");
    }
}

class Proxy implements Subject{
    private Subject subject;
    public Proxy(Subject subject){
        this.subject = subject;
    }
    public void request(){
        System.out.println("PreProcess");
        subject.request();
        System.out.println("PostProcess");
    }
}
```

目标对象(RealSubject)以及代理对象（Proxy）都实现了主题接口（Subject）。在代理对象（Proxy）中，通过构造函数传入目标对象(RealSubject)，然后重写主题接口（Subject）的request()方法，在该方法中调用目标对象(RealSubject)的request()方法，并可以添加一些额外的处理工作在目标对象(RealSubject)的request()方法的前后。

代理模式的好处：

假如有这样的需求，要在某些模块方法调用前后加上一些统一的前后处理操作，比如在添加购物车、修改订单等操作前后统一加上登陆验证与日志记录处理，该怎样实现？首先想到最简单的就是直接修改源码，在对应模块的对应方法前后添加操作。如果模块很多，你会发现，修改源码不仅非常麻烦、难以维护，而且会使代码显得十分臃肿。

这时候就轮到代理模式上场了，它可以在被调用方法前后加上自己的操作，而不需要更改被调用类的源码，大大地降低了模块之间的耦合性，体现了极大的优势。

静态代理比较简单，上面的简单实例就是静态代理的应用方式，下面介绍本篇文章的主题：动态代理。

二、Java反射机制与动态代理

动态代理的思路和上述思路一致，下面主要讲解如何实现。

1、动态代理介绍

动态代理是指在运行时动态生成代理类。即，代理类的字节码将在运行时生成并载入当前代理的 ClassLoader。与静态处理类相比，动态类有诸多好处。

①不需要为(RealSubject)写一个形式上完全一样的封装类，假如主题接口（Subject）中的方法很多，为每一个接口写一个代理方法也很麻烦。如果接口有变动，则目标对象和代理类都要修改，不利于系统维护；

②使用一些动态代理的生成方法甚至可以在运行时制定代理类的执行逻辑，从而大大提升系统的灵活性。

2、动态代理涉及的主要类

主要涉及两个类，这两个类都是java.lang.reflect包下的类，内部主要通过反射来实现的。

java.lang.reflect.Proxy:这是生成代理类的主类，通过 Proxy 类生成的代理类都继承了 Proxy 类。
Proxy提供了用户创建动态代理类和代理对象的静态方法，它是所有动态代理类的父类。

java.lang.reflect.InvocationHandler:这里称他为"调用处理器"，它是一个接口。当调用动态代理类中的方法时，将会直接转接到执行自定义的InvocationHandler中的invoke()方法。即我们动态生成的代理类需要完成的具体内容需要自己定义一个类，而这个类必须实现 InvocationHandler 接口，通过重写invoke()方法来执行具体内容。

Proxy提供了如下两个方法来创建动态代理类和动态代理实例。

> static Class<?> getProxyClass(ClassLoader loader, Class<?>... interfaces) 返回代理类的java.lang.Class对象。第一个参数是类加载器对象（即哪个类加载器来加载这个代理类到 JVM 的方法区），第二个参数是接口（表明你这个代理类需要实现哪些接口），第三个参数是调用处理器类实例（指定代理类中具体要干什么），该代理类将实现interfaces所指定的所有接口，执行代理对象的每个方法时都会被替换执行InvocationHandler对象的invoke方法。
>
> static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h) 返回代理类实例。参数与上述方法一致。

对应上述两种方法创建动态代理对象的方式：

```
        //创建一个InvocationHandler对象
        InvocationHandler handler = new MyInvocationHandler(.args..);
        //使用Proxy生成一个动态代理类
        Class proxyClass = Proxy.getProxyClass(RealSubject.class.getClassLoader(),RealSubject.class.getInterfaces(), handler);
        //获取proxyClass类中一个带InvocationHandler参数的构造器
        Constructor constructor = proxyClass.getConstructor(InvocationHandler.class);
        //调用constructor的newInstance方法来创建动态实例
        RealSubject real = (RealSubject)constructor.newInstance(handler);
```

```
        //创建一个InvocationHandler对象
        InvocationHandler handler = new MyInvocationHandler(.args..);
        //使用Proxy直接生成一个动态代理对象
        RealSubject real =Proxy.newProxyInstance(RealSubject.class.getClassLoader(),RealSubject.class.getInterfaces(), handler);
```

newProxyInstance这个方法实际上做了两件事：第一，创建了一个新的类【代理类】，这个类实现了**Class[] interfaces**中的所有接口，并通过你指定的**ClassLoader**将生成的类的字节码加载到**JVM**中，创建**Class**对象；第二，以你传入的**InvocationHandler**作为参数创建一个代理类的实例并返回。

Proxy 类还有一些静态方法，比如：

`InvocationHandler getInvocationHandler(Object proxy):` 获得代理对象对应的调用处理器对象。

`Class getProxyClass(ClassLoader loader, Class[] interfaces):` 根据类加载器和实现的接口获得代理类。

InvocationHandler 接口中有方法：

`invoke(Object proxy, Method method, Object[] args)`
这个函数是在代理对象调用任何一个方法时都会调用的，方法不同会导致第二个参数method不同，第一个参数是代理对象（表示哪个代理对象调用了method方法），第二个参数是 Method 对象（表示哪个方法被调用了），第三个参数是指定调用方法的参数。

3、动态代理模式的简单实现

```
public class DynamicProxyDemo {
    public static void main(String[] args) {
        //1.创建目标对象
        RealSubject realSubject = new RealSubject();
        //2.创建调用处理器对象
        ProxyHandler handler = new ProxyHandler(realSubject);

       //3.动态生成代理对象
        Subject proxySubject = (Subject)Proxy.newProxyInstance(R
```

```
ealSubject.class.getClassLoader(),
                                                        RealSubj
ect.class.getInterfaces(), handler);
        //4.通过代理对象调用方法
        proxySubject.request();
    }
}

/**
 * 主题接口
 */
interface Subject{
    void request();
}

/**
 * 目标对象类
 */
class RealSubject implements Subject{
    public void request(){
        System.out.println("====RealSubject Request====");
    }
}
/**
 * 代理类的调用处理器
 */
class ProxyHandler implements InvocationHandler{
    private Subject subject;
    public ProxyHandler(Subject subject){
        this.subject = subject;
    }
    @Override
    public Object invoke(Object proxy, Method method, Object[] a
rgs)
            throws Throwable {
        //定义预处理的工作，当然你也可以根据 method 的不同进行不同的预处
理工作
        System.out.println("====before====");
       //调用RealSubject中的方法
        Object result = method.invoke(subject, args);
```

```
            System.out.println("====after====");
            return result;
        }
    }
```

可以看到，我们通过newProxyInstance就产生了一个Subject 的实例，即代理类的实例，然后就可以通过Subject .request()，就会调用InvocationHandler中的invoke()方法，传入方法Method对象，以及调用方法的参数，通过Method.invoke调用RealSubject中的方法的request()方法。同时可以在InvocationHandler中的invoke()方法加入其他执行逻辑。

一、泛型和Class类

从JDK 1.5 后，Java中引入泛型机制，Class类也增加了泛型功能，从而允许使用泛型来限制Class类，例如：String.class的类型实际上是Class<String>。如果Class对应的类暂时未知，则使用Class<?>(?是通配符)。通过反射中使用泛型，可以避免使用反射生成的对象需要强制类型转换。

泛型的好处众多，最主要的一点就是避免类型转换，防止出现ClassCastException，即类型转换异常。以下面程序为例：

```
public class ObjectFactory {
    public static Object getInstance(String name){
        try {
            //创建指定类对应的Class对象
            Class cls = Class.forName(name);
            //返回使用该Class对象创建的实例
            return cls.newInstance();
        } catch (ClassNotFoundException | InstantiationException
 | IllegalAccessException e) {
            e.printStackTrace();
            return null;
        }
    }

}
```

上面程序是个工厂类，通过指定的字符串创建Class对象并创建一个类的实例对象返回。但是这个对象的类型是Object对象，取出实例后需要强制类型转换。
如下例：

```
Date date = (Date) ObjectFactory.getInstance("java.util.Date");
```

或者如下：

```
String string = (String) ObjectFactory.getInstance("java.util.Da
te");
```

上面代码在编译时不会有任何问题，但是运行时将抛出ClassCastException异常，因为程序试图将一个Date对象转换成String对象。

但是泛型的出现后，就可以避免这种情况。

```
public class ObjectFactory {
    public static <T> T getInstance(Class<T> cls) {
        try {
            // 返回使用该Class对象创建的实例
            return cls.newInstance();
        } catch (InstantiationException | IllegalAccessException
 e) {
            e.printStackTrace();
            return null;
        }
    }

}
```

在上面程序的getInstance()方法中传入一个Class<T>参数，这是一个泛型化的Class对象，调用该Class对象的newInstance()方法将返回一个T对象。

```
String instance = ObjectFactory.getInstance(String.class);
```

通过传入 `String.class` 便知道T代表String，所以返回的对象是String类型的，避免强制类型转换。

当然Class类引入泛型的好处不止这一点，在以后的实际应用中会更加能体会到。

二、使用反射来获取泛型信息

通过指定类对应的 Class 对象，可以获得该类里包含的所有 Field，不管该 Field 是使用 private 修饰，还是使用 public 修饰。获得了 Field 对象后，就可以很容易地获得该 Field 的数据类型，即使用如下代码即可获得指定 Field 的类型。

```
// 获取 Field 对象 f 的类型
Class<?> a = f.getType();
```

但这种方式只对普通类型的 Field 有效。如果该 Field 的类型是有泛型限制的类型，如 Map<String, Integer> 类型，则不能准确地得到该 Field 的泛型参数。

为了获得指定 Field 的泛型类型，应先使用如下方法来获取指定 Field 的类型。

```
// 获得 Field 实例的泛型类型
Type type = f.getGenericType();
```

然后将 Type 对象强制类型转换为 ParameterizedType 对象，ParameterizedType 代表被参数化的类型，也就是增加了泛型限制的类型。ParameterizedType 类提供了如下两个方法。

getRawType()：返回没有泛型信息的原始类型。

getActualTypeArguments()：返回泛型参数的类型。

下面是一个获取泛型类型的完整程序。

```
public class GenericTest
{
    private Map<String , Integer> score;
    public static void main(String[] args)
        throws Exception
    {
        Class<GenericTest> clazz = GenericTest.class;
        Field f = clazz.getDeclaredField("score");
        // 直接使用getType()取出Field类型只对普通类型的Field有效
        Class<?> a = f.getType();
        // 下面将看到仅输出java.util.Map
        System.out.println("score的类型是:" + a);
        // 获得Field实例f的泛型类型
        Type gType = f.getGenericType();
        // 如果gType类型是ParameterizedType对象
        if(gType instanceof ParameterizedType)
        {
            // 强制类型转换
            ParameterizedType pType = (ParameterizedType)gType;
            // 获取原始类型
            Type rType = pType.getRawType();
            System.out.println("原始类型是：" + rType);
            // 取得泛型类型的泛型参数
            Type[] tArgs = pType.getActualTypeArguments();
            System.out.println("泛型类型是:");
            for (int i = 0; i < tArgs.length; i++)
            {
                System.out.println("第" + i + "个泛型类型是：" + tA
rgs[i]);
            }
        }
        else
        {
            System.out.println("获取泛型类型出错！");
        }
    }
}
```

输出结果：

> score 的类型是: interface java.util.Map
> 原始类型是: interface java.util.Map
> 泛型类型是:
> 第 0 个泛型类型是: class java.lang.String
> 第 1 个泛型类型是：class java.lang.Integer

从上面的运行结果可以看出，直接使用 Field 的 getType() 方法只能获取普通类型的 Field 的数据类型：对于增加了泛型参数的类型的 Field，应该使用 getGenericType() 方法来取得其类型。

Type 也是 java.lang.reflect 包下的一个接口，该接口代表所有类型的公共高级接口，Class 是 Type 接口的实现类。Type 包括原始类型、参数化类型、数组类型、类型变量和基本类型等。

一、元数据

要想理解注解（Annotation）的作用，就要先理解Java中元数据的概念。

1.元数据概念

元数据是关于数据的数据。在编程语言上下文中，元数据是添加到程序元素如方法、字段、类和包上的额外信息。对数据进行说明描述的数据。

2.元数据的作用

一般来说，元数据可以用于创建文档（根据程序元素上的注释创建文档），跟踪代码中的依赖性（可声明方法是重载，依赖父类的方法），执行编译时检查（可声明是否编译期检测），代码分析。
如下：
1） 编写文档：通过代码里标识的元数据生成文档
2）代码分析：通过代码里标识的元数据对代码进行分析
3）编译检查：通过代码里标识的元数据让编译器能实现基本的编译检查

3.Java平台元数据

注解Annotation就是java平台的元数据，是 J2SE5.0新增加的功能，该机制允许在Java 代码中添加自定义注释，并允许通过反射（reflection），以编程方式访问元数据注释。通过提供为程序元素（类、方法等）附加额外数据的标准方法，元数据功能具有简化和改进许多应用程序开发领域的潜在能力，其中包括配置管理、框架实现和代码生成。

二、注解（Annotation）

1.注解（Annotation）的概念

注解(Annotation)在JDK1.5之后增加的一个新特性，注解的引入意义很大，有很多非常有名的框架，比如Hibernate、Spring等框架中都大量使用注解。注解作为程序的元数据嵌入到程序。注解可以被解析工具或编译工具解析。

关于注解（Annotation）的作用，其实就是上述元数据的作用。

注意：**Annotation**能被用来为程序元素（类、方法、成员变量等）设置元素据。**Annotaion**不影响程序代码的执行，无论增加、删除**Annotation**，代码都始终如一地执行。如果希望让程序中的**Annotation**起一定的作用，只有通过解析工具或编译工具对**Annotation**中的信息进行解析和处理。

2.内建注解

Java提供了多种内建的注解，下面接下几个比较常用的注解：@Override、@Deprecated、@SuppressWarnings以及@FunctionalInterface这4个注解。内建注解主要实现了元数据的第二个作用：编译检查。

@Override

用途：用于告知编译器，我们需要覆写超类的当前方法。如果某个方法带有该注解但并没有覆写超类相应的方法，则编译器会生成一条错误信息。如果父类没有这个要覆写的方法，则编译器也会生成一条错误信息。

@Override可适用元素为方法，仅仅保留在java源文件中。

@Deprecated

用途：使用这个注解，用于告知编译器，某一程序元素(比如方法，成员变量)不建议使用了（即过时了）。

例如：

Person类中的info()方法使用 `@Deprecated` 表示该方法过时了。

```
public class Person {
    @Deprecated
    public void info(){

    }
}
```

调用info()方法会编译器会出现警告，告知该方法已过时。

```
public class Main {
    public static void main(String[] args) {
        Person person = new Person();
        person.info();
    }
}
```

注解类型分析：`@Deprecated` 可适合用于除注解类型声明之外的所有元素，保留时长为运行时。

@SuppressWarnings

用途：用于告知编译器忽略特定的警告信息，例在泛型中使用原生数据类型，编译器会发出警告，当使用该注解后，则不会发出警告。

注解类型分析：`@SuppressWarnings` 可适合用于除注解类型声明和包名之外的所有元素，仅仅保留在java源文件中。

该注解有方法value（）,可支持多个字符串参数，用户指定忽略哪种警告，例如：

```
@SupressWarning(value={"uncheck","deprecation"})
```

参数	含义
deprecation	使用了过时的类或方法时的警告
unchecked	执行了未检查的转换时的警告
fallthrough	当Switch程序块进入进入下一个case而没有Break时的警告
path	在类路径、源文件路径等有不存在路径时的警告
serial	当可序列化的类缺少serialVersionUID定义时的警告
finally	任意finally子句不能正常完成时的警告
all	以上所有情况的警告

@FunctionalInterface

用途：用户告知编译器，检查这个接口，保证该接口是函数式接口，即只能包含一个抽象方法，否则就会编译出错。

注解类型分析：`@FunctionalInterface` 可适合用于注解类型声明，保留时长为运行时。

3.元Annotation

JDK除了在java.lang提供了上述内建注解外，还在java.lang。annotation包下提供了6个Meta Annotation(元Annotataion)，其中有5个元Annotation都用于修饰其他的Annotation定义。其中@Repeatable专门用户定义Java 8 新增的可重复注解。

我们先介绍其中4个常用的修饰其他Annotation的元Annotation。在此之前，我们先了解如何自定义Annotation。

当一个接口直接继承**java.lang.annotation.Annotation**接口时，仍是接口，而并非注解。要想自定义注解类型，只能通过**@interface**关键字的方式，其实通过该方式会隐含地继承**.Annotation**接口。

@Documented

`@Documented` 用户指定被该元Annotation修饰的Annotation类将会被javadoc工具提取成文档，如果定义Annotation类时使用了 `@Documented` 修饰，则所有使用该Annotation修饰的程序元素的API文档中将会包含该Annotation说明。

例如：

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(value={CONSTRUCTOR, FIELD, LOCAL_VARIABLE, METHOD, PACKA
GE, PARAMETER, TYPE})
public @interface Deprecated {
}
```

定义 `@Deprecated` 时使用了 `@Documented` ，则任何元素使用@Deprecated修饰时，在生成API文档时，将会包含 `@Deprecated` 的说明
以下是String的一个过时的构造方法：

```
@Deprecated
public String(byte[] ascii,int hibyte,int offset, int count)
```

该注解实现了元数据的第一个功能：编写文档。

@Inherited

`@Inherited` 指定被它修饰的Annotation将具有继承性——如果某个类使用了@Xxx注解（定义该Annotation时使用了 `@Inherited` 修饰）修饰，则其子类将自动被@Xxx修饰。

@Retention

`@Retention` ：表示该注解类型的注解保留的时长。当注解类型声明中没有 `@Retention` 元注解，则默认保留策略为RetentionPolicy.CLASS。关于保留策略(RetentionPolicy)是枚举类型，共定义3种保留策略，如下表：

RetentionPolicy	含义
SOURCE	仅存在Java源文件，经过编译器后便丢弃相应的注解
CLASS	存在Java源文件，以及经编译器后生成的Class字节码文件，但在运行时VM不再保留注释
RUNTIME	存在源文件、编译生成的Class字节码文件，以及保留在运行时VM中，可通过反射性地读取注解

@Target

`@Target` ：表示该注解类型的所适用的程序元素类型。当注解类型声明中没有 `@Target` 元注解，则默认为可适用所有的程序元素。如果存在指定的 `@Target` 元注解，则编译器强制实施相应的使用限制。关于程序元素(ElementType)是枚举类型，共定义8种程序元素，如下表：

ElementType	含义
ANNOTATION_TYPE	注解类型声明
CONSTRUCTOR	构造方法声明
FIELD	字段声明（包括枚举常量）
LOCAL_VARIABLE	局部变量声明
METHOD	方法声明
PACKAGE	包声明
PARAMETER	参数声明
TYPE	类、接口（包括注解类型）或枚举声明

三、自定义注解（Annotation）

创建自定义注解，与创建接口有几分相似，但注解需要以@开头。

```
@Documented
@Target(ElementType.METHOD)
@Inherited
@Retention(RetentionPolicy.RUNTIME)
public @interface MyAnnotataion{
    String name();
    String website() default "hello";
    int revision() default 1;
}
```

自定义注解中定义成员变量的规则：

其定义是以无形参的方法形式来声明的。即：
注解方法不带参数，比如name()，website()；
注解方法返回值类型：基本类型、String、Enums、Annotation以及前面这些类型的数组类型
注解方法可有默认值，比如default "hello"，默认website="hello"

当然注解中也可以不存在成员变量，在使用解析注解进行操作时，仅以是否包含该注解来进行操作。当注解中有成员变量时，若没有默认值，需要在使用注解时，指定成员变量的值。

```
public class AnnotationDemo {
    @AuthorAnno(name="lvr", website="hello", revision=1)
    public static void main(String[] args) {
        System.out.println("I am main method");
    }

    @SuppressWarnings({ "unchecked", "deprecation" })
    @AuthorAnno(name="lvr", website="hello", revision=2)
    public void demo(){
        System.out.println("I am demo method");
    }
}
```

由于该注解的保留策略为 `RetentionPolicy.RUNTIME` ，故可在运行期通过反射机制来使用，否则无法通过反射机制来获取。这时候注解实现的就是元数据的第二个作用：代码分析。

下面来具体介绍如何通过反射机制来进行注解解析。

四、注解解析

接下来，通过反射技术来解析自定义注解。关于反射类位于包java.lang.reflect，其中有一个接口AnnotatedElement，该接口主要有如下几个实现类：Class，Constructor，Field，Method，Package。除此之外，该接口定义了注释相关的几个核心方法，如下：

返回值	方法	解释
T	getAnnotation(Class annotationClass)	当存在该元素的指定类型注解，则返回相应注释，否则返回null
Annotation[]	getAnnotations()	返回此元素上存在的所有注解
Annotation[]	getDeclaredAnnotations()	返回直接存在于此元素上的所有注解。
boolean	isAnnotationPresent(Class<? extends Annotation> annotationClass)	当存在该元素的指定类型注解，则返回true，否则返回false

因此，当获取了某个类的Class对象，然后获取其Field,Method等对象，通过上述4个方法提取其中的注解，然后获得注解的详细信息。

```
public class AnnotationParser {
    public static void main(String[] args) throws SecurityExcept
ion, ClassNotFoundException {
        String clazz = "com.lvr.annotation.AnnotationDemo";
        Method[]  demoMethod = AnnotationParser.class
                .getClassLoader().loadClass(clazz).getMethods();

        for (Method method : demoMethod) {
            if (method.isAnnotationPresent(MyAnnotataion.class))
 {
                 MyAnnotataion annotationInfo = method.getAnnota
tion(MyAnnotataion.class);
                 System.out.println("method: "+ method);
                 System.out.println("name= "+ annotationInfo.nam
e() +
                         " , website= "+ annotationInfo.website(
)
                        + " , revision= "+annotationInfo.revisio
n());
            }
        }
    }
}
```

以上仅是一个示例，其实可以根据拿到的注解信息做更多有意义的事。

一、字符与字节

在Java中有输入、输出两种IO流，每种输入、输出流又分为字节流和字符流两大类。关于字节，我们在学习8大基本数据类型中都有了解，每个字节(byte)有8bit组成，每种数据类型又几个字节组成等。关于字符，我们可能知道代表一个汉字或者英文字母。

但是字节与字符之间的关系是怎样的？

Java采用unicode编码，2个字节来表示一个字符，这点与C语言中不同，C语言中采用ASCII，在大多数系统中，一个字符通常占1个字节，但是在0~127整数之间的字符映射，unicode向下兼容ASCII。而Java采用unicode来表示字符，一个中文或英文字符的unicode编码都占2个字节。但如果采用其他编码方式，一个字符占用的字节数则各不相同。可能有点晕，举个例子解释下。

例如：Java中的String类是按照unicode进行编码的，当使用String(byte[] bytes, String encoding)构造字符串时，encoding所指的是bytes中的数据是按照那种方式编码的，而不是最后产生的String是什么编码方式，换句话说，是让系统把bytes中的数据由encoding编码方式转换成unicode编码。如果不指明，bytes的编码方式将由jdk根据操作系统决定。

`getBytes(String charsetName)` 使用指定的编码方式将此String编码为 byte 序列，并将结果存储到一个新的 byte 数组中。如果不指定将使用操作系统默认的编码方式，我的电脑默认的是GBK编码。

```
public class Hel {
    public static void main(String[] args){
        String str = "你好hello";
            int byte_len = str.getBytes().length;
            int len = str.length();
            System.out.println("字节长度为：" + byte_len);
        System.out.println("字符长度为：" + len);
        System.out.println("系统默认编码方式：" + System.getPropert
y("file.encoding"));
       }
}
```

输出结果

> 字节长度为：9
> 字符长度为：7
> 系统默认编码方式：GBK

这是因为：在 GB 2312 编码或 GBK 编码中，一个英文字母字符存储需要1个字节，一个汉字字符存储需要2个字节。 在UTF-8编码中，一个英文字母字符存储需要1个字节，一个汉字字符储存需要3到4个字节。在UTF-16编码中，一个英文字母字符存储需要2个字节，一个汉字字符储存需要3到4个字节（Unicode扩展区的一些汉字存储需要4个字节）。在UTF-32编码中，世界上任何字符的存储都需要4个字节。

简单来讲，一个字符表示一个汉字或英文字母，具体字符与字节之间的大小比例视编码情况而定。有时候读取的数据是乱码，就是因为编码方式不一致，需要进行转换，然后再按照**unicode**进行编码。

二、File类

File类是java.io包下代表与平台无关的文件和目录，也就是说，如果希望在程序中操作文件和目录，都可以通过File类来完成。

①构造函数

```
//构造函数File(String pathname)
File f1 =new File("c:\\abc\\1.txt");
//File(String parent,String child)
File f2 =new File("c:\\abc","2.txt");
//File(File parent,String child)
File f3 =new File("c:"+File.separator+"abc");//separator 跨平台分隔符
File f4 =new File(f3,"3.txt");
System.out.println(f1);//c:\abc\1.txt
```

路径分隔符：
windows： "/" "\" 都可以
linux/unix： "/"
注意:如果windows选择用"\"做分割符的话,那么请记得替换成"\",因为Java中"\"代表

转义字符

所以推荐都使用"/"，也可以直接使用代码 `File.separator` ，表示跨平台分隔符。

路径：

相对路径：

./表示当前路径

../表示上一级路径

其中当前路径：默认情况下，java.io 包中的类总是根据当前用户目录来分析相对路径名。此目录由系统属性 user.dir 指定，通常是 Java 虚拟机的调用目录。”

绝对路径：

绝对路径名是完整的路径名，不需要任何其他信息就可以定位自身表示的文件

②创建与删除方法

```
//如果文件存在返回false，否则返回true并且创建文件
boolean createNewFile();
//创建一个File对象所对应的目录，成功返回true，否则false。且File对象必须为路径而不是文件。只会创建最后一级目录，如果上级目录不存在就抛异常。
boolean mkdir();
//创建一个File对象所对应的目录，成功返回true，否则false。且File对象必须为路径而不是文件。创建多级目录，创建路径中所有不存在的目录
boolean mkdirs()    ;
//如果文件存在返回true并且删除文件，否则返回false
boolean delete();
//在虚拟机终止时，删除File对象所表示的文件或目录。
void deleteOnExit();
```

③判断方法

```
boolean canExecute()    ;//判断文件是否可执行
boolean canRead();//判断文件是否可读
boolean canWrite();//判断文件是否可写
boolean exists();//判断文件是否存在
boolean isDirectory();//判断是否是目录
boolean isFile();//判断是否是文件
boolean isHidden();//判断是否是隐藏文件或隐藏目录
boolean isAbsolute();//判断是否是绝对路径 文件不存在也能判断
```

③获取方法

```
String getName();//返回文件或者是目录的名称
String getPath();//返回路径
String getAbsolutePath();//返回绝对路径
String getParent();//返回父目录，如果没有父目录则返回null
long lastModified();//返回最后一次修改的时间
long length();//返回文件的长度
File[] listRoots();// 列出所有的根目录（Window中就是所有系统的盘符）
String[] list() ;//返回一个字符串数组，给定路径下的文件或目录名称字符串
String[] list(FilenameFilter filter);//返回满足过滤器要求的一个字符串数组
File[]  listFiles();//返回一个文件对象数组，给定路径下文件或目录
File[] listFiles(FilenameFilter filter);//返回满足过滤器要求的一个文件对象数组
```

其中包含了一个重要的接口FileNameFilter，该接口是个文件过滤器，包含了一个 `accept(File dir,String name)` 方法，该方法依次对指定File的所有子目录或者文件进行迭代，按照指定条件，进行过滤，过滤出满足条件的所有文件。

```
        // 文件过滤
        File[] files = file.listFiles(new FilenameFilter() {
            @Override
            public boolean accept(File file, String filename) {
                return filename.endsWith(".mp3");
            }
        });
```

file目录下的所有子文件如果满足后缀是.mp3的条件的文件都会被过滤出来。

一、IO流的概念

Java的IO流是实现输入/输出的基础，它可以方便地实现数据的输入/输出操作，在Java中把不同的输入/输出源抽象表述为"流"。流是一组有顺序的，有起点和终点的字节集合，是对数据传输的总称或抽象。即数据在两设备间的传输称为流，流的本质是数据传输，根据数据传输特性将流抽象为各种类，方便更直观的进行数据操作。

流有输入和输出，输入时是流从数据源流向程序。输出时是流从程序传向数据源，而数据源可以是内存，文件，网络或程序等。

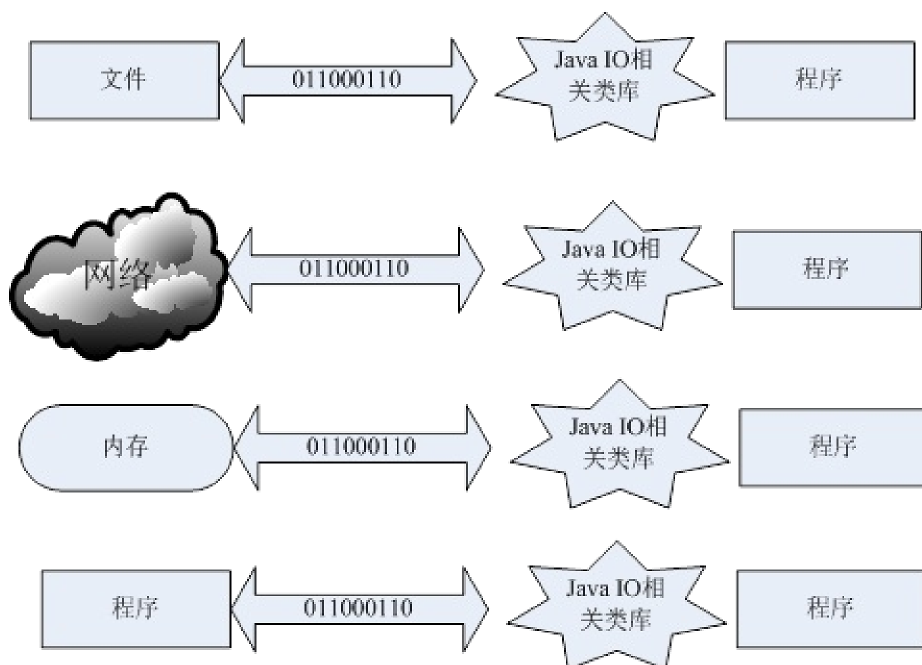
二、IO流的分类

1.输入流和输出流

根据数据流向不同分为：输入流和输出流。

> 输入流:只能从中读取数据，而不能向其写入数据。
> 输出流：只能向其写入数据，而不能从中读取数据。

如下如所示：对程序而言，向右的箭头，表示输入，向左的箭头，表示输出。



2.字节流和字符流

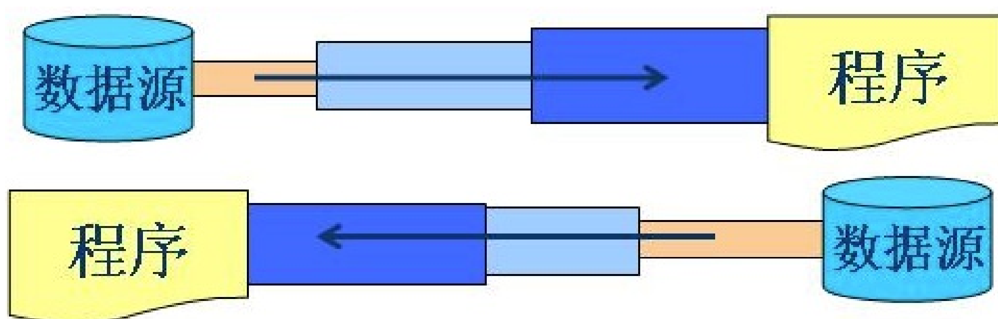
字节流和字符流和用法几乎完全一样，区别在于字节流和字符流所操作的数据单元不同。
字符流的由来： 因为数据编码的不同，而有了对字符进行高效操作的流对象。本质其实就是基于字节流读取时，去查了指定的码表。字节流和字符流的区别：
（1）读写单位不同：字节流以字节（8bit）为单位，字符流以字符为单位，根据码表映射字符，一次可能读多个字节。
（2）处理对象不同：字节流能处理所有类型的数据（如图片、avi等），而字符流只能处理字符类型的数据。

只要是处理纯文本数据，就优先考虑使用字符流。 除此之外都使用字节流。

3.节点流和处理流

按照流的角色来分，可以分为节点流和处理流。
可以从/向一个特定的IO设备（如磁盘、网络）读/写数据的流，称为节点流，节点流也被成为低级流。
处理流是对一个已存在的流进行连接或封装，通过封装后的流来实现数据读/写功能，处理流也被称为高级流。

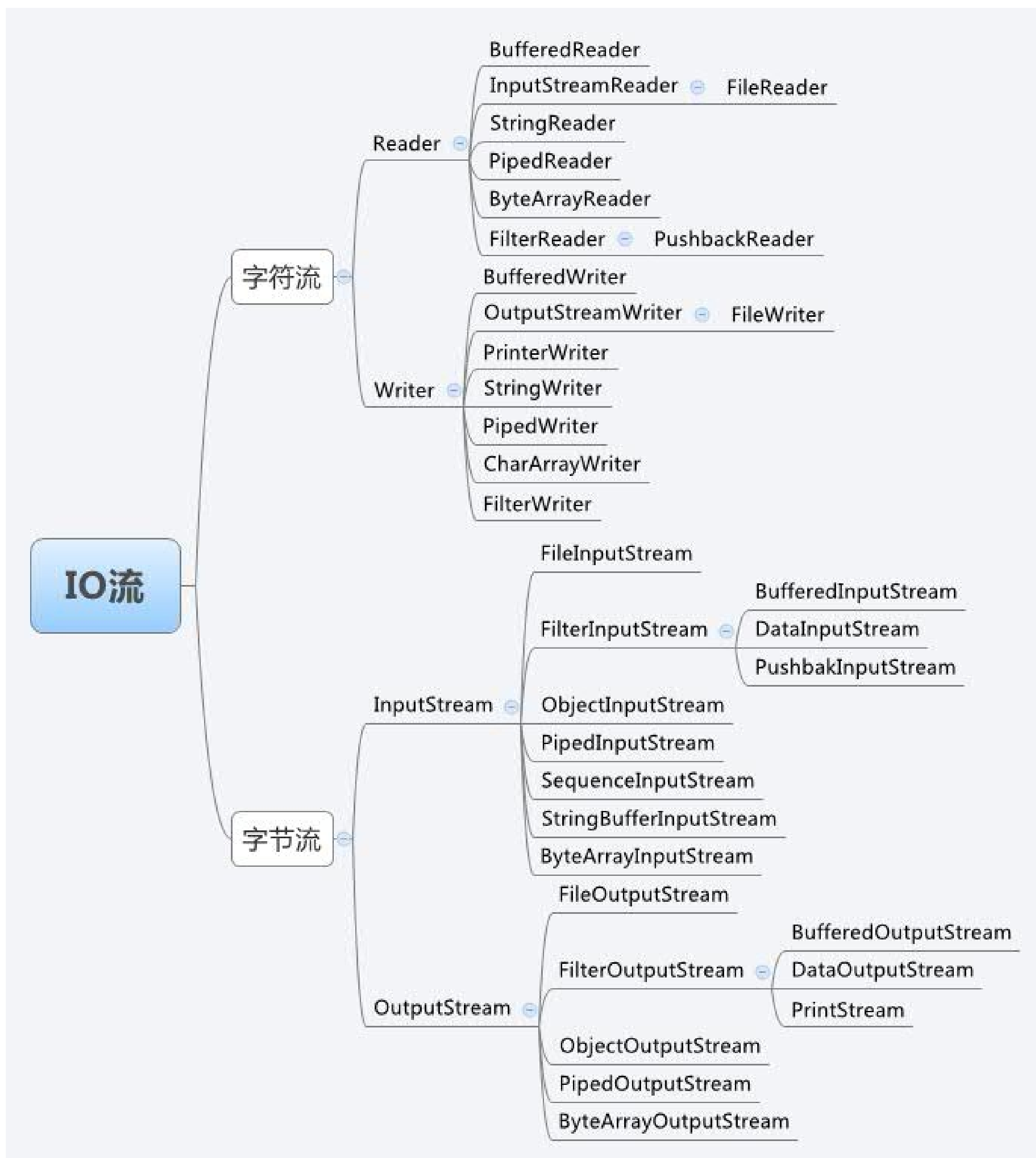
```
//节点流，直接传入的参数是IO设备
FileInputStream fis = new FileInputStream("test.txt");
//处理流，直接传入的参数是流对象
BufferedInputStream bis = new BufferedInputStream(fis);
```



当使用处理流进行输入/输出时，程序并不会直接连接到实际的数据源，没有和实际的输入/输出节点连接。使用处理流的一个明显好处是，只要使用相同的处理流，程序就可以采用完全相同的输入/输出代码来访问不同的数据源，随着处理流所包装节点流的变化，程序实际所访问的数据源也相应地发生变化。
实际上，Java使用处理流来包装节点流是一种典型的装饰器设计模式，通过使用处理流来包装不同的节点流，既可以消除不同节点流的实现差异，也可以提供更方便的方法来完成输入/输出功能。

三、IO流的四大基类

根据流的流向以及操作的数据单元不同，将流分为了四种类型，每种类型对应一种抽象基类。这四种抽象基类分别为：InputStream,Reader,OutputStream以及Writer。四种基类下，对应不同的实现类，具有不同的特性。在这些实现类中，又可以分为节点流和处理流。下面就是整个由着四大基类支撑下，整个IO流的框架图。



InputStream,Reader,OutputStream以及Writer，这四大抽象基类，本身并不能创建实例来执行输入/输出，但它们将成为所有输入/输出流的模版，所以它们的方法是所有输入/输出流都可以使用的方法。类似于集合中的Collection接口。

1.InputStream

InputStream 是所有的输入字节流的父类，它是一个抽象类，主要包含三个方法：

```
//读取一个字节并以整数的形式返回(0~255),如果返回-1已到输入流的末尾。
int read() ；
//读取一系列字节并存储到一个数组buffer，返回实际读取的字节数，如果读取前已到输入流的末尾返回-1。
int read(byte[] buffer) ；
//读取length个字节并存储到一个字节数组buffer，从off位置开始存,最多len，返回实际读取的字节数，如果读取前以到输入流的末尾返回-1。
int read(byte[] buffer, int off, int len) ；
```

2.Reader

Reader 是所有的输入字符流的父类，它是一个抽象类，主要包含三个方法：

```
//读取一个字符并以整数的形式返回(0~255),如果返回-1已到输入流的末尾。
int read() ；
//读取一系列字符并存储到一个数组buffer，返回实际读取的字符数，如果读取前已到输入流的末尾返回-1。
int read(char[] cbuf) ；
//读取length个字符,并存储到一个数组buffer，从off位置开始存,最多读取len，返回实际读取的字符数，如果读取前以到输入流的末尾返回-1。
int read(char[] cbuf, int off, int len)
```

对比InputStream和Reader所提供的方法，就不难发现两个基类的功能基本一样的，只不过读取的数据单元不同。

在执行完流操作后，要调用 `close()` 方法来关系输入流，因为程序里打开的**IO**资源不属于内存资源，垃圾回收机制无法回收该资源，所以应该显式关闭文件**IO**资源。

除此之外，InputStream和Reader还支持如下方法来移动流中的指针位置：

```
//在此输入流中标记当前的位置
//readlimit - 在标记位置失效前可以读取字节的最大限制。
void mark(int readlimit)
// 测试此输入流是否支持 mark 方法
boolean markSupported()
// 跳过和丢弃此输入流中数据的 n 个字节/字符
long skip(long n)
//将此流重新定位到最后一次对此输入流调用 mark 方法时的位置
void reset()
```

3.OutputStream

OutputStream 是所有的输出字节流的父类，它是一个抽象类，主要包含如下四个方法：

```
//向输出流中写入一个字节数据,该字节数据为参数b的低8位。
void write(int b) ;
//将一个字节类型的数组中的数据写入输出流。
void write(byte[] b);
//将一个字节类型的数组中的从指定位置（off）开始的,len个字节写入到输出流。
void write(byte[] b, int off, int len);
//将输出流中缓冲的数据全部写出到目的地。
void flush();
```

4.Writer

Writer 是所有的输出字符流的父类，它是一个抽象类,主要包含如下六个方法：

```
//向输出流中写入一个字符数据,该字节数据为参数b的低16位。
void write(int c);
//将一个字符类型的数组中的数据写入输出流，
void write(char[] cbuf)
//将一个字符类型的数组中的从指定位置（offset）开始的,length个字符写入到输出流。
void write(char[] cbuf, int offset, int length);
//将一个字符串中的字符写入到输出流。
void write(String string);
//将一个字符串从offset开始的length个字符写入到输出流。
void write(String string, int offset, int length);
//将输出流中缓冲的数据全部写出到目的地。
void flush()
```

可以看出，Writer比OutputStream多出两个方法，主要是支持写入字符和字符串类型的数据。

使用**Java**的**IO**流执行输出时，不要忘记关闭输出流，关闭输出流除了可以保证流的物理资源被回收之外，还能将输出流缓冲区的数据**flush**到物理节点里（因为在执行**close()**方法之前，自动执行输出流的**flush()**方法）

以上内容就是整个IO流的框架介绍。

一、RandomAccessFile简介

RandomAccessFile既可以读取文件内容，也可以向文件输出数据。同时，RandomAccessFile支持“随机访问”的方式，程序快可以直接跳转到文件的任意地方来读写数据。

由于RandomAccessFile可以自由访问文件的任意位置，所以如果需要访问文件的部分内容，而不是把文件从头读到尾，使用**RandomAccessFile**将是更好的选择。

与OutputStream、Writer等输出流不同的是，RandomAccessFile允许自由定义文件记录指针，RandomAccessFile可以不从开始的地方开始输出，因此RandomAccessFile可以向已存在的文件后追加内容。如果程序需要向已存在的文件后追加内容，则应该使用**RandomAccessFile**。

RandomAccessFile的方法虽然多，但它有一个最大的局限，就是只能读写文件，不能读写其他IO节点。

RandomAccessFile的一个重要使用场景就是网络请求中的多线程下载及断点续传。

二、RandomAccessFile中的方法

1.RandomAccessFile的构造函数

RandomAccessFile类有两个构造函数，其实这两个构造函数基本相同，只不过是指定文件的形式不同——一个需要使用String参数来指定文件名，一个使用File参数来指定文件本身。除此之外，创建RandomAccessFile对象时还需要指定一个mode参数，该参数指定RandomAccessFile的访问模式，一共有4种模式。

> **"r":** 以只读方式打开。调用结果对象的任何 write 方法都将导致抛出 IOException。
> **"rw":** 打开以便读取和写入。
> **"rws":** 打开以便读取和写入。相对于 "rw"，"rws" 还要求对“文件的内容”或“元数据”的每个更新都同步写入到基础存储设备。
> **"rwd" :** 打开以便读取和写入，相对于 "rw"，"rwd" 还要求对“文件的内容”的每个更新都同步写入到基础存储设备。

2.RandomAccessFile的重要方法

RandomAccessFile既可以读文件，也可以写文件，所以类似于InputStream的read()方法，以及类似于OutputStream的write()方法，RandomAccessFile都具备。除此之外，RandomAccessFile具备两个特有的方法，来支持其随机访问的特性。

RandomAccessFile对象包含了一个记录指针，用以标识当前读写处的位置，当程序新创建一个RandomAccessFile对象时，该对象的文件指针记录位于文件头（也就是0处），当读/写了n个字节后，文件记录指针将会后移n个字节。除此之外，RandomAccessFile还可以自由移动该记录指针。下面就是RandomAccessFile具有的两个特殊方法，来操作记录指针，实现随机访问：

> long getFilePointer( )：返回文件记录指针的当前位置
> void seek(long pos)：将文件指针定位到pos位置

三、RandomAccessFile的使用

利用RandomAccessFile实现文件的多线程下载，即多线程下载一个文件时，将文件分成几块，每块用不同的线程进行下载。下面是一个利用多线程在写文件时的例子，其中预先分配文件所需要的空间，然后在所分配的空间中进行分块，然后写入：

```
/**
 * 测试利用多线程进行文件的写操作
 */
public class Test {

    public static void main(String[] args) throws Exception {
        // 预分配文件所占的磁盘空间，磁盘中会创建一个指定大小的文件
        RandomAccessFile raf = new RandomAccessFile("D://abc.txt"
, "rw");
        raf.setLength(1024*1024); // 预分配 1M 的文件空间
        raf.close();

        // 所要写入的文件内容
        String s1 = "第一个字符串";
        String s2 = "第二个字符串";
        String s3 = "第三个字符串";
        String s4 = "第四个字符串";
        String s5 = "第五个字符串";
```

```
        // 利用多线程同时写入一个文件
        new FileWriteThread(1024*1,s1.getBytes()).start(); // 从
文件的1024字节之后开始写入数据
        new FileWriteThread(1024*2,s2.getBytes()).start(); // 从
文件的2048字节之后开始写入数据
        new FileWriteThread(1024*3,s3.getBytes()).start(); // 从
文件的3072字节之后开始写入数据
        new FileWriteThread(1024*4,s4.getBytes()).start(); // 从
文件的4096字节之后开始写入数据
        new FileWriteThread(1024*5,s5.getBytes()).start(); // 从
文件的5120字节之后开始写入数据
    }

    // 利用线程在文件的指定位置写入指定数据
    static class FileWriteThread extends Thread{
        private int skip;
        private byte[] content;

        public FileWriteThread(int skip,byte[] content){
            this.skip = skip;
            this.content = content;
        }

        public void run(){
            RandomAccessFile raf = null;
            try {
                raf = new RandomAccessFile("D://abc.txt", "rw");

                raf.seek(skip);
                raf.write(content);
            } catch (FileNotFoundException e) {
                e.printStackTrace();
            } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            } finally {
                try {
                    raf.close();
                } catch (Exception e) {
```

```
                    }
                }
            }
        }

    }
```

当**RandomAccessFile**向指定文件中插入内容时，将会覆盖掉原有内容。如果不想覆盖掉，则需要将原有内容先读取出来，然后先把插入内容插入后再把原有内容追加到插入内容后。

一、NIO的概念

Java NIO(New IO)是一个可以替代标准Java IO API的IO API(从Java1.4开始)，Java NIO提供了与标准IO不同的IO工作方式。

所以Java NIO是一种新式的IO标准，与之间的普通IO的工作方式不同。标准的IO基于字节流和字符流进行操作的，而NIO是基于通道(Channel)和缓冲区(Buffer)进行操作，数据总是从通道读取到缓冲区中，或者从缓冲区写入通道也类似。

由上面的定义就说明**NIO**是一种新型的**IO**，但**NIO**不仅仅就是等于**Non-blocking IO**（非阻塞**IO**），**NIO**中有实现非阻塞**IO**的具体类，但不代表**NIO**就是**Non-blocking IO**（非阻塞**IO**）。

Java NIO 由以下几个核心部分组成：

- Buffer
- Channel
- Selector

传统的IO操作面向数据流，意味着每次从流中读一个或多个字节，直至完成，数据没有被缓存在任何地方。NIO操作面向缓冲区，数据从Channel读取到Buffer缓冲区，随后在Buffer中处理数据。

二、Buffer的使用

利用**Buffer**读写数据，通常遵循四个步骤：

1. 把数据写入buffer；
2. 调用flip；
3. 从Buffer中读取数据；
4. 调用buffer.clear()

当写入数据到buffer中时，buffer会记录已经写入的数据大小。当需要读数据时，通过flip()方法把buffer从写模式调整为读模式；在读模式下，可以读取所有已经写入的数据。

当读取完数据后，需要清空buffer，以满足后续写入操作。清空buffer有两种方式：调用clear()，一旦读完Buffer中的数据，需要让Buffer准备好再次被写入，clear会恢复状态值，但不会擦除数据。

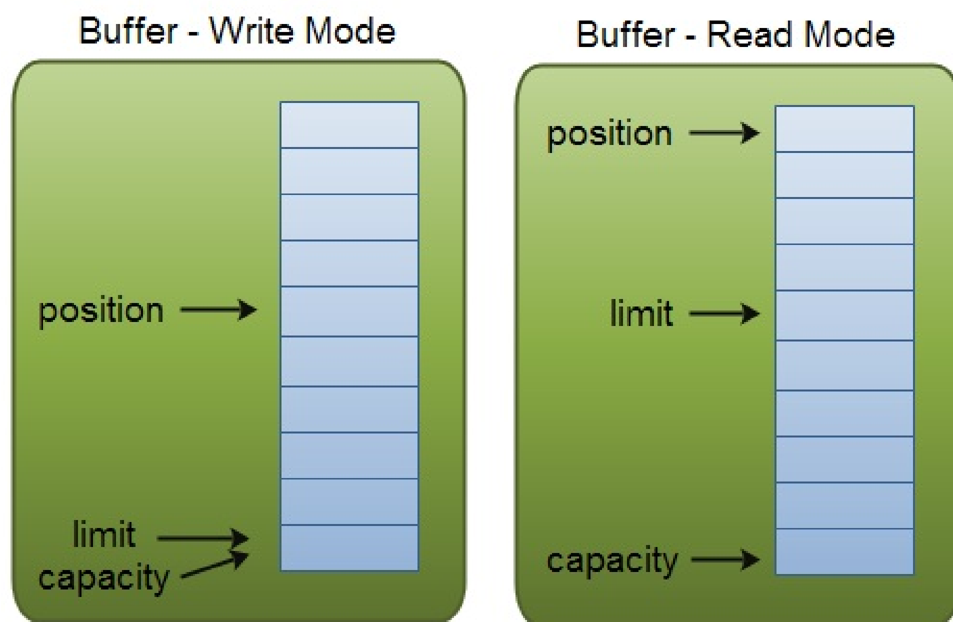
Buffer的容量，位置，上限（**Buffer Capacity, Position and Limit**）

buffer缓冲区实质上就是一块内存，用于写入数据，也供后续再次读取数据。这块内存被NIO Buffer管理，并提供一系列的方法用于更简单的操作这块内存。

一个Buffer有三个属性是必须掌握的，分别是：

- capacity容量
- position位置
- limit限制

position和limit的具体含义取决于当前buffer的模式。capacity在两种模式下都表示容量。
下面有张示例图，描诉了不同模式下position和limit的含义：



容量（**Capacity**）

作为一块内存，buffer有一个固定的大小，叫做capacity容量。也就是最多只能写入容量值得字节，整形等数据。一旦buffer写满了就需要清空已读数据以便下次继续写入新的数据。

位置（**Position**）

当写入数据到Buffer的时候需要中一个确定的位置开始，默认初始化时这个位置position为0，一旦写入了数据比如一个字节，整形数据，那么position的值就会指向数据之后的一个单元，position最大可以到capacity-1。

当从Buffer读取数据时，也需要从一个确定的位置开始。buffer从写入模式变为读取模式时，position会归零，每次读取后，position向后移动。

上限（**Limit**）
在写模式，limit的含义是我们所能写入的最大数据量。它等同于buffer的容量。
一旦切换到读模式，limit则代表我们所能读取的最大数据量，他的值等同于写模式下position的位置。
数据读取的上限时buffer中已有的数据，也就是limit的位置（原position所指的位置）。

分配一个**Buffer**（**Allocating a Buffer**）

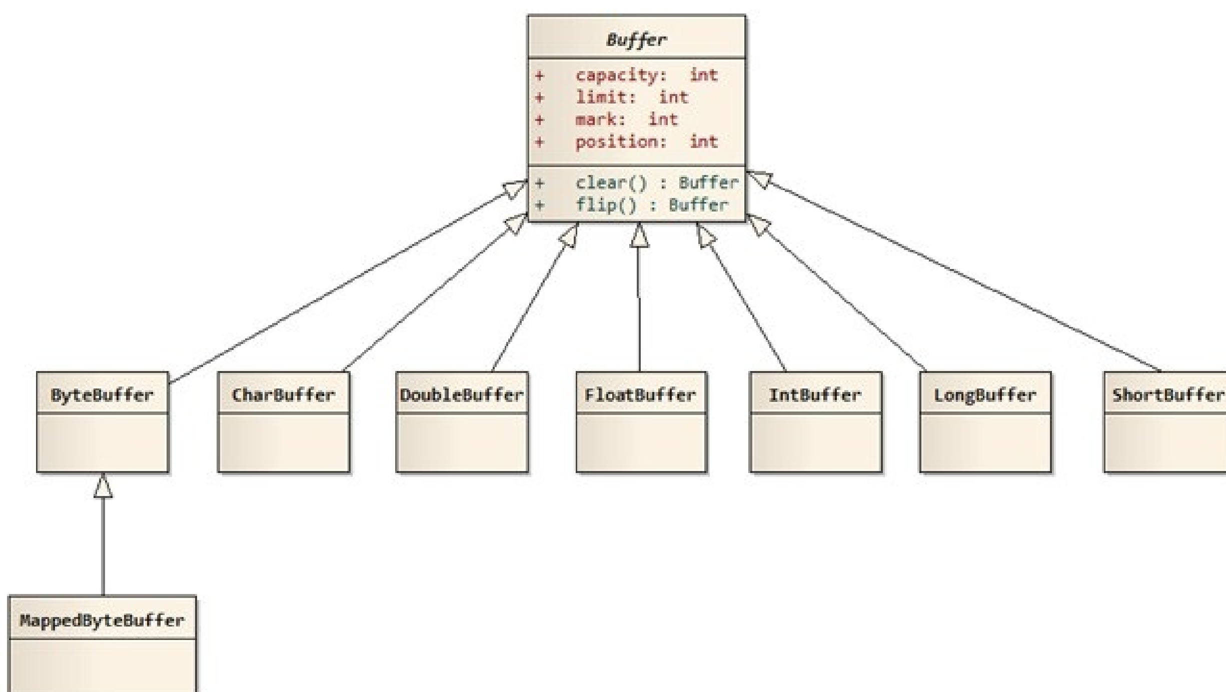
为了获取一个Buffer对象，你必须先分配。每个Buffer实现类都有一个allocate()方法用于分配内存。下面看一个实例,开辟一个48字节大小的buffer：

```
ByteBuffer buf = ByteBuffer.allocate(48);
```

开辟一个1024个字符的CharBuffer：

```
CharBuffer buf = CharBuffer.allocate(1024);
```

Buffer的实现类

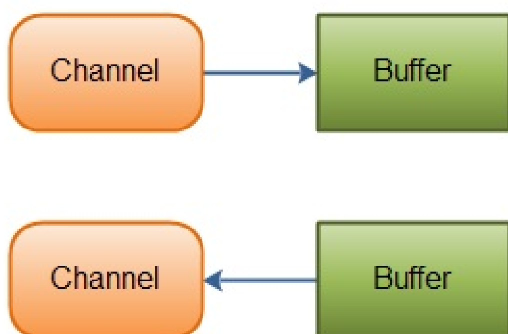


其中MappedByteBuffer比较特殊。Java类库中的NIO包相对于IO 包来说有一个新功能是内存映射文件，日常编程中并不是经常用到，但是在处理大文件时是比较理想的提高效率的手段。其中MappedByteBuffer实现的就是内存映射文件，可以实现大文件的高效读写。 可以参考这两篇文章理解：[Java][IO]JAVA NIO之浅谈内存映射文件原理与DirectMemory，深入浅出MappedByteBuffer。

三、Channel的使用

Java NIO Channel通道和流非常相似，主要有以下几点区别：

- 通道可以读也可以写，流一般来说是单向的（只能读或者写）。
- 通道可以异步读写。
- 通道总是基于缓冲区Buffer来读写。
- 正如上面提到的，我们可以从通道中读取数据，写入到buffer；也可以中buffer内读数据，写入到通道中。下面有个示意图：



Channel的实现类有：

- FileChannel
- DatagramChannel
- SocketChannel
- ServerSocketChannel

还有一些异步IO类，后面有介绍。

FileChannel用于文件的数据读写。 DatagramChannel用于UDP的数据读写。 SocketChannel用于TCP的数据读写。 ServerSocketChannel允许我们监听TCP链接请求，每个请求会创建会一个SocketChannel。

Channel使用实例

```
RandomAccessFile aFile = new RandomAccessFile("data/nio-data.txt"
, "rw");
    FileChannel inChannel = aFile.getChannel();

    ByteBuffer buf = ByteBuffer.allocate(48);

    int bytesRead = inChannel.read(buf);
    while (bytesRead != -1) {

      System.out.println("Read " + bytesRead);
      buf.flip();

      while(buf.hasRemaining()){
          System.out.print((char) buf.get());
      }

      buf.clear();
      bytesRead = inChannel.read(buf);
    }
    aFile.close();
```

上面介绍了NIO中的两个关键部分Buffer/Channel，对于Selector的介绍，先放一放，先介绍阻塞/非阻塞/同步/非同步的关系。

四、阻塞/非阻塞/同步/非同步的关系

为什么要介绍这四者的关系，就是因为Selector是对于多个非阻塞IO流的调度器，通过Selector来实现读写操作。所以有必要理解一下什么是阻塞/非阻塞？

本文讨论的背景是UNIX环境下的network IO。本文最重要的参考文献是Richard Stevens的“**UNIX® Network Programming Volume 1, Third Edition: The Sockets Networking** ”，6.2节“**I/O Models** ”，Stevens在这节中详细说明了各种IO的特点和区别。

Stevens在文章中一共比较了五种IO Model：

- blocking IO
- nonblocking IO
- IO multiplexing
- signal driven IO
- asynchronous IO。

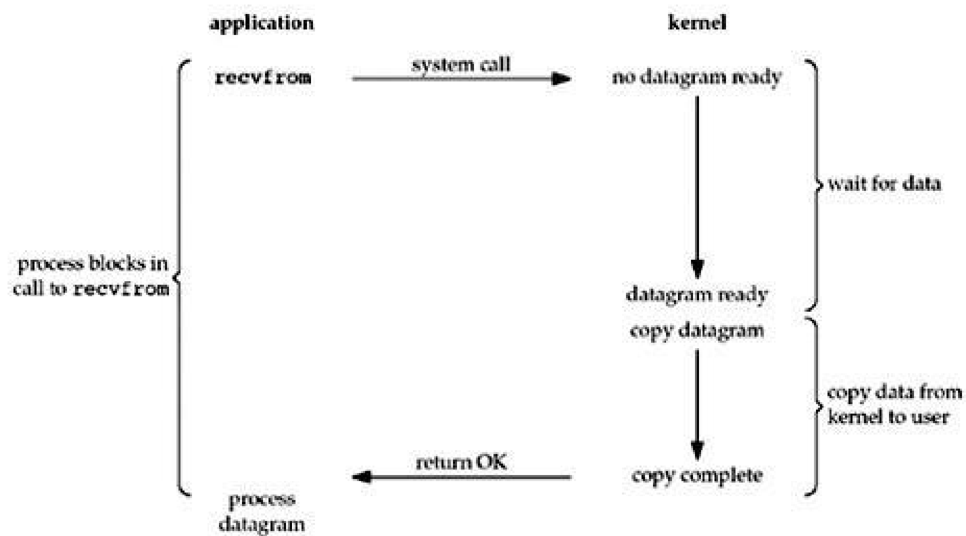
由于signal driven IO在实际中并不常用，所以我这只提及剩下的四种IO Model。再说一下IO发生时涉及的对象和步骤。对于一个network IO (这里我们以read举例)，它会涉及到两个系统对象，一个是调用这个IO的process (or thread)，另一个就是系统内核(kernel)。

当一个read操作发生时，它会经历两个阶段：
1 等待数据准备 **(Waiting for the data to be ready)**
2 将数据从内核拷贝到进程中 **(Copying the data from the kernel to the process)**

记住这两点很重要，因为这些IO Model的区别就是在两个阶段上各有不同的情况。

blocking IO

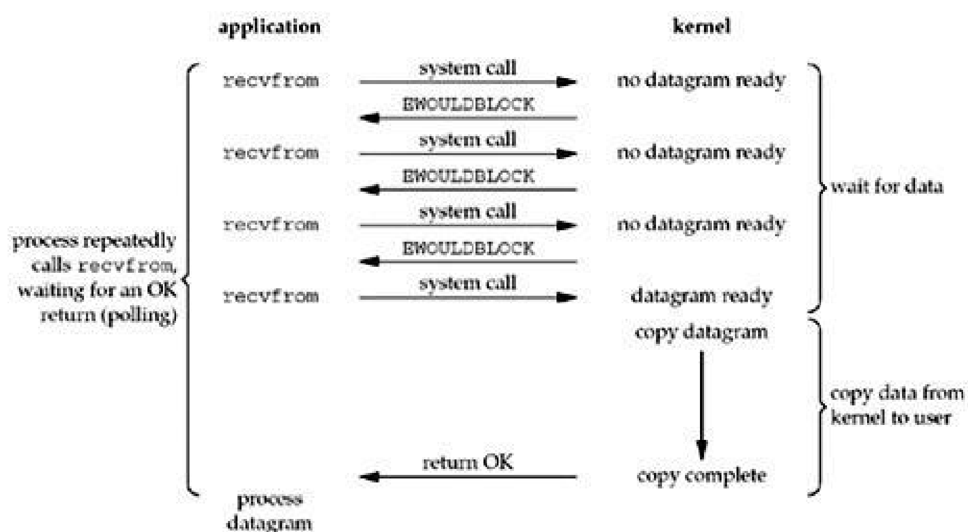
在UNIX中，默认情况下所有的socket都是blocking，一个典型的读操作流程大概是这样：



当用户进程调用了recvfrom这个系统调用，kernel就开始了IO的第一个阶段：准备数据。对于network io来说，很多时候数据在一开始还没有到达（比如，还没有收到一个完整的UDP包），这个时候kernel就要等待足够的数据到来。而在用户进程这边，整个进程会被阻塞。当kernel一直等到数据准备好了，它就会将数据从kernel中拷贝到用户内存，然后kernel返回结果，用户进程才解除block的状态，重新运行起来。所以，**blocking IO**的特点就是在**IO**执行的两个阶段都被**block**了。

non-blocking IO

UNIX下，可以通过设置socket使其变为non-blocking。当对一个non-blocking socket执行读操作时，流程是这个样子：

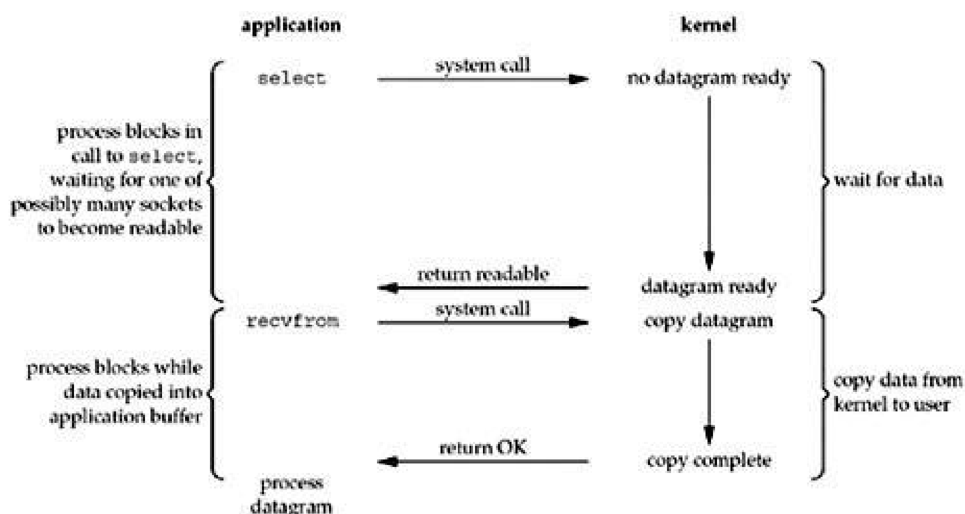


从图中可以看出，当用户进程发出read操作时，如果kernel中的数据还没有准备好，那么它并不会block用户进程，而是立刻返回一个error。从用户进程角度讲 ，它发起一个read操作后，并不需要等待，而是马上就得到了一个结果。用户进程判断结果是一个error时，它就知道数据还没有准备好，于是它可以再次发送read操作。一旦**kernel**中的数据准备好了，并且又再次收到了用户进程的**system call**，那么它马上就将数据拷贝到了用户内存，然后返回。所以，用户进程其实是需要不断的主动询问**kernel**数据好了没有。

IO multiplexing

IO multiplexing这个词可能有点陌生，但是如果我说select，epoll，大概就都能明白了。有些地方也称这种IO方式为event driven IO。我们都知道，select/epoll的好处就在于单个process就可以同时处理多个网络连接的IO。它的基本原理就是

select/epoll这个function会不断的轮询所负责的所有socket，当某个socket有数据到达了，就通知用户进程。它的流程如图：



当用户进程调用了select，那么整个进程会被block，而同时，kernel会"监视"所有select负责的socket，当任何一个socket中的数据准备好了，select就会返回。这个时候用户进程再调用read操作，将数据从kernel拷贝到用户进程。

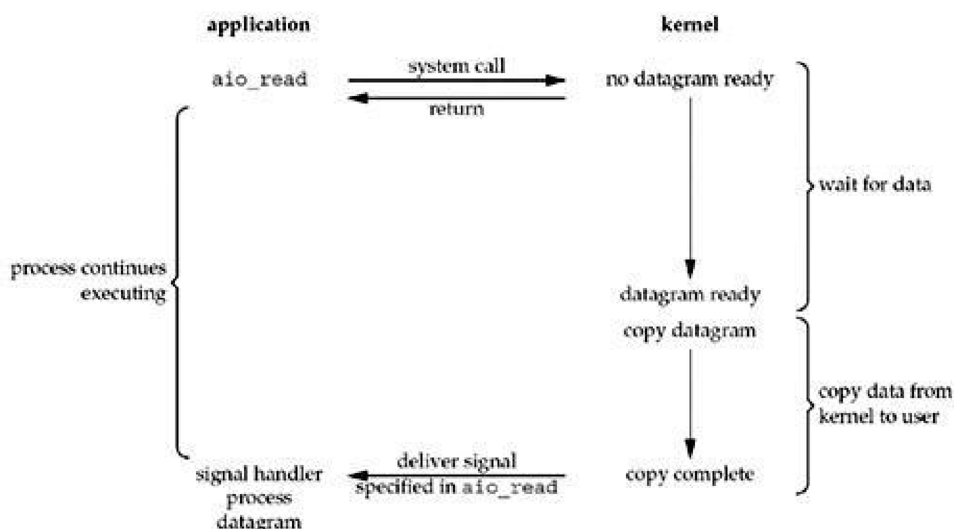
这个图和blocking IO的图其实并没有太大的不同，事实上，还更差一些。因为这里需要使用两个system call (select 和 recvfrom)，而blocking IO只调用了一个system call (recvfrom)。但是，用select的优势在于它可以同时处理多个connection。（多说一句。所以，如果处理的连接数不是很高的话，使用select/epoll的web server不一定比使用multi-threading + blocking IO的web server性能更好，可能延迟还更大。select/epoll的优势并不是对于单个连接能处理得更快，而是在于能处理更多的连接。）

在IO multiplexing Model中，实际中，对于每一个**socket**，一般都设置成为**non-blocking**，但是，如上图所示，整个用户的process其实是一直被block的。只不过**process**是被**select**这个函数**block**，而不是被**socket IO**给**block**。

Asynchronous I/O

UNIX下的asynchronous IO其实用得很少。先看一下它的流程：

Figure 6.5. Asynchronous I/O model.



用户进程发起**read**操作之后，立刻就可以开始去做其它的事。 而另一方面，从kernel的角度，当它受到一个asynchronous read之后，首先它会立刻返回，所以不会对用户进程产生任何block。然后，**kernel**会等待数据准备完成，然后将数据拷贝到用户内存，当这一切都完成之后，**kernel**会给用户进程发送一个**signal**，告诉它**read**操作完成了。

到目前为止，已经将四个IO Model都介绍完了。现在回过头来回答最初的那几个问题：

blocking和**non-blocking**的区别在哪，**synchronous IO**和**asynchronous IO**的区别在哪？

先回答最简单的这个：blocking vs non-blocking。前面的介绍中其实已经很明确的说明了这两者的区别。调用blocking IO会一直block住对应的进程直到操作完成，而non-blocking IO在kernel还准备数据的情况下会立刻返回。

在说明synchronous IO和asynchronous IO的区别之前，需要先给出两者的定义。Stevens给出的定义（其实是POSIX的定义）是这样子的：
A synchronous I/O operation causes the requesting process to be blocked until that I/O operationcompletes; An asynchronous I/O operation does not cause the requesting process to be blocked;

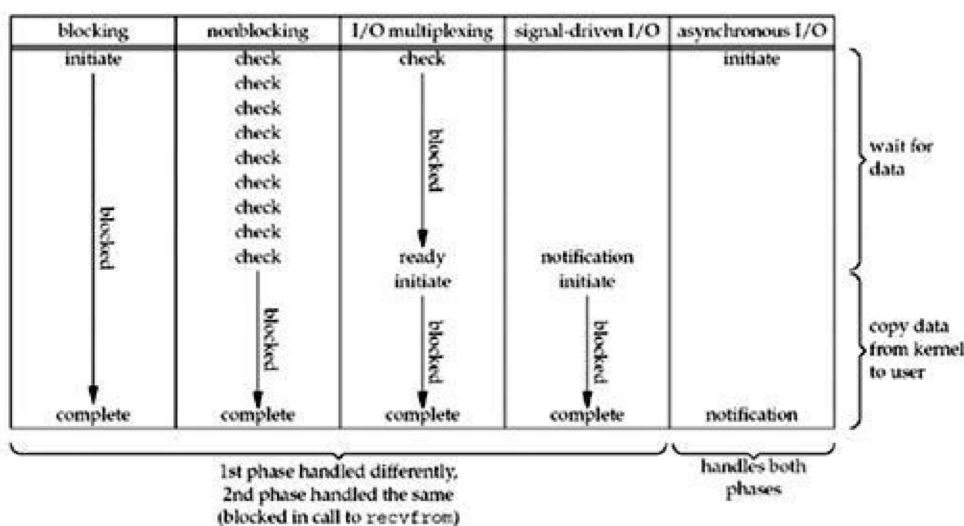
两者的区别就在于synchronous IO做"IO operation"的时候会将process阻塞。

按照这个定义，之前所述的**blocking IO**，**non-blocking IO**，**IO multiplexing**都属于**synchronous IO**。

有人可能会说，non-blocking IO并没有被block啊。这里有个非常“狡猾”的地方，定义中所指的”IO operation”是指真实的IO操作，就是例子中的recvfrom这个system call。non-blocking IO在执行recvfrom这个system call的时候，如果kernel的数据没有准备好，这时候不会block进程。但是，当kernel中数据准备好的时候，recvfrom会将数据从kernel拷贝到用户内存中，这个时候进程是被block了，在这段时间内，进程是被block的。而asynchronous IO则不一样，当进程发起IO 操作之后，就直接返回再也不理睬了，直到kernel发送一个信号，告诉进程说IO完成。在这整个过程中，进程完全没有被block。

各个IO Model的比较如图所示：

Figure 6.6. Comparison of the five I/O models.



经过上面的介绍，会发现non-blocking IO和asynchronous IO的区别还是很明显的。在non-blocking IO中，虽然进程大部分时间都不会被block，但是它仍然要求进程去主动的check，并且当数据准备完成以后，也需要进程主动的再次调用recvfrom来将数据拷贝到用户内存。而asynchronous IO则完全不同。它就像是用户进程将整个IO操作交给了他人（kernel）完成，然后他人做完后发信号通知。在此期间，用户进程不需要去检查IO操作的状态，也不需要主动的去拷贝数据。

五、NIO中的blocking IO/nonblocking IO/IO multiplexing/asynchronous IO

上面讲完了IO中的几种模式，虽然是基于UNIX环境下，具体操作系统的知识个人认识很浅，下面就说下自己的个人理解，不对的地方欢迎指正。

首先，标准的IO显然属于blocking IO。

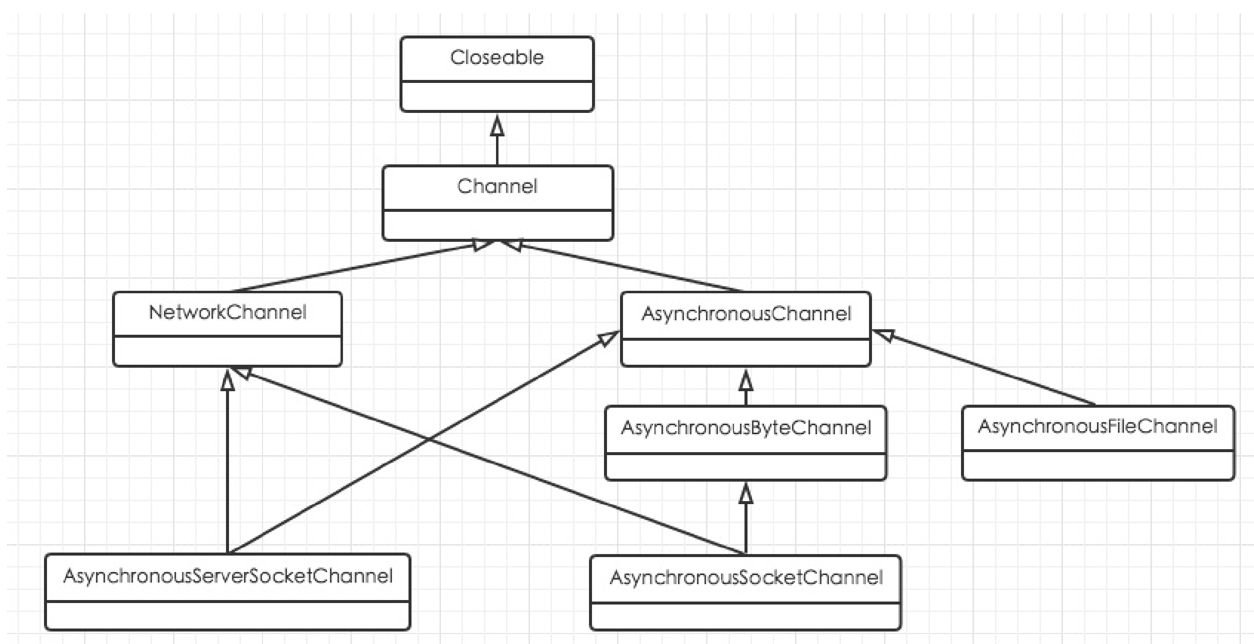
其次，NIO中的实现了SelectableChannel类的对象，可以通过如下方法设置是否支持非阻塞模式：

> SelectableChannel configureBlocking(boolean block)：调整此通道的阻塞模式。

如果为 true，则此通道将被置于阻塞模式；如果为 false，则此通道将被置于非阻塞模式
设置为false的NIO类将是nonblocking IO。

再其次，通过Selector监听实现多个NIO对象的读写操作，显然属于IO multiplexing。关于Selector，其负责调度多个非阻塞式IO，当有其感兴趣的读写操作到来时，再执行相应的操作。Selector执行select()方法来进行轮询查找是否到来了读写操作，这个过程是阻塞的，具体详细使用下面介绍。

最后，在Java 7中增加了asynchronous IO，具体结构和实现类框架如下：



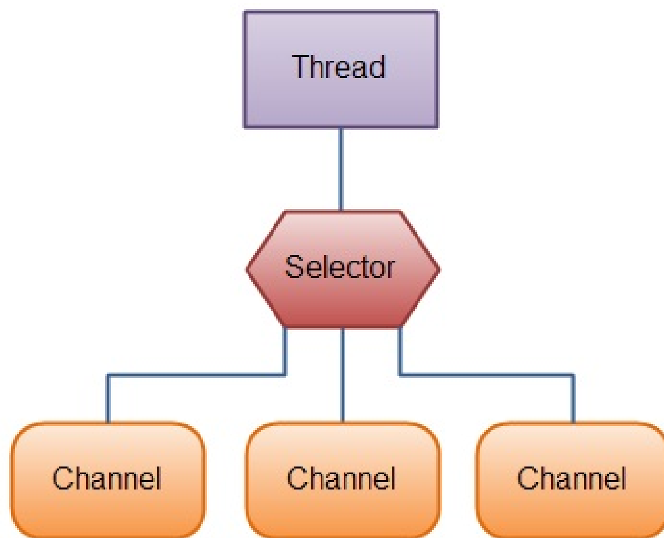
篇幅有限，具体使用可以看这篇文章：Java 学习之路 之 基于TCP协议的网络编程（八十二）。

六、Selector使用

Selector是Java NIO中的一个组件，用于检查一个或多个NIO Channel的状态是否处于可读、可写。如此可以实现单线程管理多个channels,也就是可以管理多个网络链接。

通过上面的了解我们知道Selector是一种IO multiplexing的情况。

下面这幅图描述了单线程处理三个channel的情况：



创建**Selector(Creating a Selector)**。创建一个**Selector**可以通过**Selector.open()**方法：

```
Selector selector = Selector.open();
```

注册**Channel**到**Selector**上：

```
channel.configureBlocking(false);
SelectionKey key = channel.register(selector, SelectionKey.OP_RE
AD);
```

Channel必须是非阻塞的。上面对IO multiplexing的图解中可以看出。所以FileChannel不适用Selector，因为FileChannel不能切换为非阻塞模式。Socket channel可以正常使用。

注意**register**的第二个参数，这个参数是一个“关注集合”，代表我们关注的**channel**状态，有四种基础类型可供监听：

- Connect
- Accept
- Read
- Write

一个**channel**触发了一个事件也可视作该事件处于就绪状态。

因此当channel与server连接成功后，那么就是“Connetct”状态。server channel接收请求连接时处于“Accept”状态。channel有数据可读时处于“Read”状态。channel可以进行数据写入时处于“Writer”状态。当注册到Selector的所有Channel注册完后，调用Selector的select()方法，将会不断轮询检查是否有以上设置的状态产生，如果产生便会加入到SelectionKey集合中，进行后续操作。

上述的四种就绪状态用SelectionKey中的常量表示如下：

- SelectionKey.OP_CONNECT
- SelectionKey.OP_ACCEPT
- SelectionKey.OP_READ
- SelectionKey.OP_WRITE

如果对多个事件感兴趣可利用位的或运算结合多个常量，比如：

int interestSet = SelectionKey.OP_READ | SelectionKey.OP_WRITE;

从Selector中选择channel(Selecting Channels via a Selector)

一旦我们向Selector注册了一个或多个channel后，就可以调用select来获取channel。select方法会返回所有处于就绪状态的channel。

select方法具体如下：

> int select()
> int select(long timeout)
> int selectNow()

select()方法在返回channel之前处于阻塞状态。 select(long timeout)和select做的事一样，不过他的阻塞有一个超时限制。

selectNow()不会阻塞，根据当前状态立刻返回合适的channel。

select()方法的返回值是一个int整形，代表有多少channel处于就绪了。也就是自上一次select后有多少channel进入就绪。

举例来说，假设第一次调用select时正好有一个channel就绪，那么返回值是1，并且对这个channel做任何处理，接着再次调用select，此时恰好又有一个新的channel就绪，那么返回值还是1，现在我们一共有两个channel处于就绪，但是在每次调用select时只有一个channel是就绪的。

selectedKeys()

在调用select并返回了有channel就绪之后，可以通过选中的key集合来获取channel，这个操作通过调用selectedKeys()方法：

```
Set<SelectionKey> selectedKeys = selector.selectedKeys();
```

遍历这些SelectionKey可以通过如下方法：

```
Set<SelectionKey> selectedKeys = selector.selectedKeys();

Iterator<SelectionKey> keyIterator = selectedKeys.iterator();

while(keyIterator.hasNext()) {

    SelectionKey key = keyIterator.next();

    if(key.isAcceptable()) {
        // a connection was accepted by a ServerSocketChannel.

    } else if (key.isConnectable()) {
        // a connection was established with a remote server.

    } else if (key.isReadable()) {
        // a channel is ready for reading

    } else if (key.isWritable()) {
        // a channel is ready for writing
    }

    keyIterator.remove();
}
```

上述循环会迭代key集合，针对每个key我们单独判断他是处于何种就绪状态。

注意keyIterater.remove()方法的调用，Selector本身并不会移除SelectionKey对象，这个操作需要我们收到执行。当下次channel处于就绪是，Selector任然会把这些key再次加入进来。

SelectionKey.channel返回的channel实例需要强转为我们实际使用的具体的channel类型，例如ServerSocketChannel或SocketChannel.

wakeUp()

由于调用select而被阻塞的线程，可以通过调用Selector.wakeup()来唤醒即便此时已然没有channel处于就绪状态。具体操作是，在另外一个线程调用wakeup，被阻塞与select方法的线程就会立刻返回。

close()

当操作Selector完毕后，需要调用close方法。close的调用会关闭Selector并使相关的SelectionKey都无效。channel本身不管被关闭。

完整的**Selector**案例

这有一个完整的案例，首先打开一个Selector,然后注册channel，最后调用select()获取感兴趣的操作：

```
Selector selector = Selector.open();

channel.configureBlocking(false);

SelectionKey key = channel.register(selector, SelectionKey.OP_RE
AD);

while(true) {

  int readyChannels = selector.select();

  if(readyChannels == 0) continue;

  Set<SelectionKey> selectedKeys = selector.selectedKeys();

  Iterator<SelectionKey> keyIterator = selectedKeys.iterator();

  while(keyIterator.hasNext()) {

    SelectionKey key = keyIterator.next();

    if(key.isAcceptable()) {
        // a connection was accepted by a ServerSocketChannel.

    } else if (key.isConnectable()) {
        // a connection was established with a remote server.

    } else if (key.isReadable()) {
        // a channel is ready for reading

    } else if (key.isWritable()) {
        // a channel is ready for writing
    }

    keyIterator.remove();
  }
}
```

时间： 2018-05-12 01:44:07

Java异常简介

Java异常是Java提供的一种识别及响应错误的一致性机制。

Java异常机制可以使程序中异常处理代码和正常业务代码分离，保证程序代码更加优雅，并提高程序健壮性。在有效使用异常的情况下，异常能清晰的回答what, where, why这3个问题：异常类型回答了“什么”被抛出，异常堆栈跟踪回答了“在哪“抛出，异常信息回答了“为什么“会抛出。

Java异常机制用到的几个关键字：**try**、**catch**、**finally**、**throw**、**throws**。

- **try** -- 用于监听。将要被监听的代码(可能抛出异常的代码)放在try语句块之内，当try语句块内发生异常时，异常就被抛出。
- **catch** -- 用于捕获异常。catch用来捕获try语句块中发生的异常。
- **finally** -- finally语句块总是会被执行。它主要用于回收在try块里打开的物力资源(如数据库连接、网络连接和磁盘文件)。只有finally块，执行完成之后，才会回来执行try或者catch块中的return或者throw语句，如果finally中使用了return或者throw等终止方法的语句，则就不会跳回执行，直接停止。
- **throw** -- 用于抛出异常。
- **throws** -- 用在方法签名中，用于声明该方法可能抛出的异常。

下面通过几个示例对这几个关键字进行简单了解。

示例一**:** 了解**try**和**catch**基本用法

```
public class Demo1 {

    public static void main(String[] args) {
        try {
            int i = 10/0;
              System.out.println("i="+i);
        } catch (ArithmeticException e) {
              System.out.println("Caught Exception");
            System.out.println("e.getMessage(): " + e.getMessage
());
            System.out.println("e.toString(): " + e.toString());

            System.out.println("e.printStackTrace():");
            e.printStackTrace();
        }
    }
}
```

运行结果：

```
Caught Exception
e.getMessage(): / by zero
e.toString(): java.lang.ArithmeticException: / by zero
e.printStackTrace():
java.lang.ArithmeticException: / by zero
    at Demo1.main(Demo1.java:6)
```

结果说明：在try语句块中有除数为0的操作，该操作会抛出java.lang.ArithmeticException异常。通过catch，对该异常进行捕获。

观察结果我们发现，并没有执行System.out.println("i="+i)。这说明try语句块发生异常之后，try语句块中的剩余内容就不会再被执行了。

示例二: 了解**finally**的基本用法

在"示例一"的基础上，我们添加finally语句。

```
public class Demo2 {

    public static void main(String[] args) {
        try {
            int i = 10/0;
              System.out.println("i="+i);
        } catch (ArithmeticException e) {
              System.out.println("Caught Exception");
            System.out.println("e.getMessage(): " + e.getMessage
());
            System.out.println("e.toString(): " + e.toString());

            System.out.println("e.printStackTrace():");
            e.printStackTrace();
        } finally {
            System.out.println("run finally");
        }
    }
}
```

运行结果：

```
Caught Exception
e.getMessage(): / by zero
e.toString(): java.lang.ArithmeticException: / by zero
e.printStackTrace():
java.lang.ArithmeticException: / by zero
    at Demo2.main(Demo2.java:6)
run finally
```

结果说明：最终执行了finally语句块。

示例三: 了解**throws**和**throw**的基本用法

throws是用于声明抛出的异常，而throw是用于抛出异常。

```
class MyException extends Exception {
    public MyException() {}
    public MyException(String msg) {
        super(msg);
    }
}

public class Demo3 {

    public static void main(String[] args) {
        try {
            test();
        } catch (MyException e) {
            System.out.println("Catch My Exception");
            e.printStackTrace();
        }
    }
    public static void test() throws MyException{
        try {
            int i = 10/0;
              System.out.println("i="+i);
        } catch (ArithmeticException e) {
            throw new MyException("This is MyException");
        }
    }
}
```

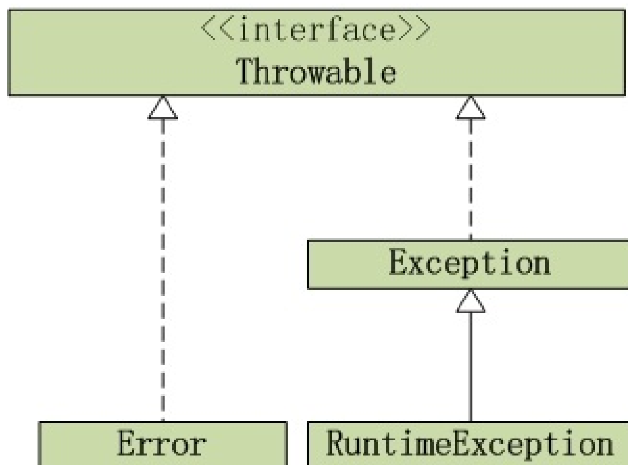
运行结果：

```
Catch My Exception
MyException: This is MyException
    at Demo3.test(Demo3.java:24)
    at Demo3.main(Demo3.java:13)
```

结果说明：

MyException是继承于Exception的子类。test()的try语句块中产生ArithmeticException异常(除数为0)，并在catch中捕获该异常；接着抛出MyException异常。main()方法对test()中抛出的MyException进行捕获处理。

Java异常框架



1. Throwable

Throwable是 Java 语言中所有错误或异常的超类。

Throwable包含两个子类: **Error** 和 **Exception**。它们通常用于指示发生了异常情况。

Throwable包含了其线程创建时线程执行堆栈的快照，它提供了printStackTrace()等接口用于获取堆栈跟踪数据等信息。

2. Exception

Exception及其子类是 Throwable 的一种形式，它指出了合理的应用程序想要捕获的条件。

3. RuntimeException

RuntimeException是那些可能在 Java 虚拟机正常运行期间抛出的异常的超类。

编译器不会检查RuntimeException异常。例如，除数为零时，抛出ArithmeticException异常。RuntimeException是ArithmeticException的超类。当代码发生除数为零的情况时，倘若既"没有通过throws声明抛出ArithmeticException异常"，也"没有通过try...catch...处理该异常"，也能通过编译。这就是我们所说的"编译器不会检查RuntimeException异常"！

如果代码会产生RuntimeException异常，则需要通过修改代码进行避免。例如，若会发生除数为零的情况，则需要通过代码避免该情况的发生！

4. Error

和Exception一样，Error也是Throwable的子类。它用于指示合理的应用程序不应该试图捕获的严重问题，大多数这样的错误都是异常条件。

和RuntimeException一样，编译器也不会检查Error。

Java将可抛出(Throwable)的结构分为三种类型：被检查的异常**(Checked Exception)**，运行时异常**(RuntimeException)**和错误**(Error)**。

(01) 运行时异常 定义: RuntimeException及其子类都被称为运行时异常。

特点: Java编译器不会检查它。也就是说，当程序中可能出现这类异常时，倘若既"没有通过throws声明抛出它"，也"没有用try-catch语句捕获它"，还是会编译通过。例如，除数为零时产生的ArithmeticException异常，数组越界时产生的IndexOutOfBoundsException异常，fail-fail机制产生的ConcurrentModificationException异常等，都属于运行时异常。

虽然Java编译器不会检查运行时异常，但是我们也可以通过throws进行声明抛出，也可以通过try-catch对它进行捕获处理。

如果产生运行时异常，则需要通过修改代码来进行避免。例如，若会发生除数为零的情况，则需要通过代码避免该情况的发生！

(02) 被检查的异常

定义: Exception类本身，以及Exception的子类中除了"运行时异常"之外的其它子类都属于被检查异常。

特点: Java编译器会检查它。此类异常，要么通过throws进行声明抛出，要么通过try-catch进行捕获处理，否则不能通过编译。例如，CloneNotSupportedException就属于被检查异常。当通过clone()接口去克隆一个对象，而该对象对应的类没有实现Cloneable接口，就会抛出CloneNotSupportedException异常。

被检查异常通常都是可以恢复的。

(03) 错误

定义: Error类及其子类。

特点: 和运行时异常一样，编译器也不会对错误进行检查。

当资源不足、约束失败、或是其它程序无法继续运行的条件发生时，就产生错误。程序本身无法修复这些错误的。例如，VirtualMachineError就属于错误。

按照Java惯例，我们是不应该是实现任何新的Error子类的！

理解抽象

abstract class和interface是Java语言中对于抽象类定义进行支持的两种机制，正是由于这两种机制的存在，才赋予了Java强大的面向对象能力。 abstract class和interface之间在对于抽象类定义的支持方面具有很大的相似性，甚至可以相互替换，因此很多开发者在进行抽象类定义时对于 abstract class和interface的选择显得比较随意。

其实，两者之间还是有很大的区别的，对于它们的选择甚至反映出对于问题领域本质的理解、对于设计意图的理解是否正确、合理。本文将对它们之间的区别进行一番剖析，试图给开发者提供一个在二者之间进行选择的依据。

语法定义理解

1. 抽象类

```
abstract class Demo ｛

    abstract void method1();

    abstract void method2();

    …

}
```

2. 接口

```
interface Demo {

    void method1();

    void method2();

    …

}
```

在abstract class方式中，Demo可以有自己的数据成员，也可以有非abstarct的成员方法，而在interface方式的实现中，Demo只能够有静态的不能被修改的数据成员（也就是必须是static final的，不过在interface中一般不定义数据成员），所有的成员方法都是abstract的。从某种意义上说，interface是一种特殊形式的abstract class。

编程角度理解

首先，abstract class在Java语言中表示的是一种继承关系，一个类只能使用一次继承关系。但是，一个类却可以实现多个interface。也许，这是Java语言的设计者在考虑Java对于多重继承的支持方面的一种折中考虑吧。

其次，在abstract class的定义中，我们可以赋予方法的默认行为。

但是在interface的定义中，方法却不能拥有默认行为，不过在JDK1.8中可以使用 `default` 关键字实现默认方法。

```
interface InterfaceA {
    default void foo() {
        System.out.println("InterfaceA foo");
    }
}
```

在 Java 8 之前，接口与其实现类之间的 耦合度 太高了（**tightly coupled**），当需要为一个接口添加方法时，所有的实现类都必须随之修改。默认方法解决了这个问题，它可以为接口添加新的方法，而不会破坏已有的接口的实现。这在 lambda 表

达式作为Java 8 语言的重要特性而出现之际，为升级旧接口且保持向后兼容（backward compatibility）提供了途径。

一般性理解

接口和抽象类的概念不一样。接口是对动作的抽象，抽象类是对根源的抽象。从设计理念上，接口反映的是 **“like-a”** 关系，抽象类反映的是 **“is-a”** 关系。 抽象类表示的是，这个对象是什么。接口表示的是，这个对象能做什么。比如，男人，女人，这两个类（如果是类的话……），他们的抽象类是人。说明，他们都是人。 人可以吃东西，狗也可以吃东西，你可以把“吃东西”定义成一个接口，然后让这些类去实现它. 所以，在高级语言上，一个类只能继承一个类（抽象类）(正如人不可能同时是生物和非生物)，但是可以实现多个接口(吃饭接口、走路接口)。

总结

1. 抽象类和接口都不能直接实例化，如果要实例化，抽象类变量必须指向实现所有抽象方法的子类对象，接口变量必须指向实现所有接口方法的类对象。
2. 抽象类要被子类继承，接口要被类实现。
3. 接口里定义的变量只能是公共的静态的常量，抽象类中的变量是普通变量。
4. 抽象类里可以没有抽象方法。
5. 接口可以被类多实现（被其他接口多继承），抽象类只能被单继承。
6. 接口中没有 `this` 指针，没有构造函数，不能拥有实例字段（实例变量）或实例方法。
7. 抽象类不能在Java 8 的 lambda 表达式中使用。

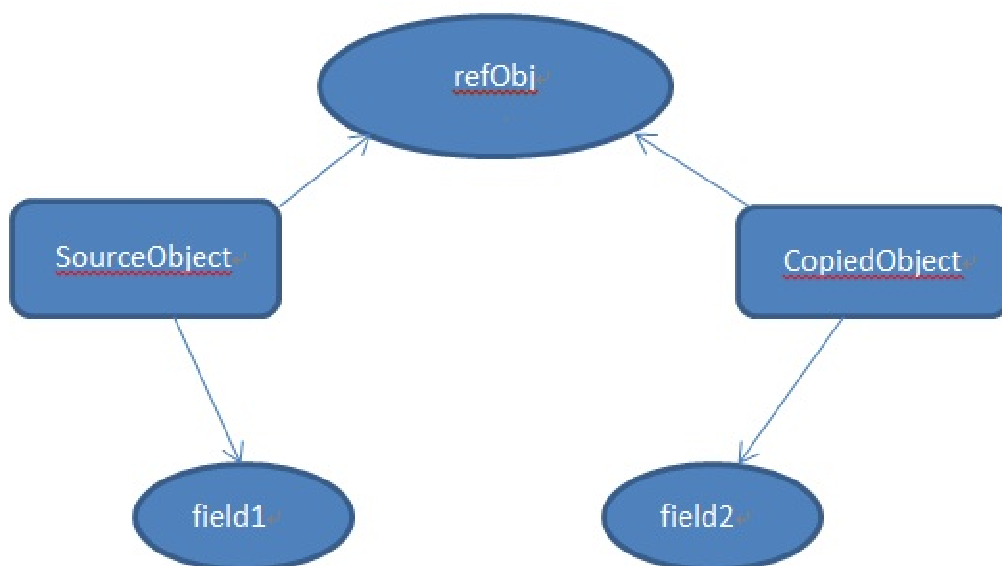
一、引言

对象拷贝(Object Copy)就是将一个对象的属性拷贝到另一个有着相同类类型的对象中去。在程序中拷贝对象是很常见的，主要是为了在新的上下文环境中复用对象的部分或全部数据。Java中有三种类型的对象拷贝：浅拷贝(Shallow Copy)、深拷贝(Deep Copy)、延迟拷贝(Lazy Copy)。

二、浅拷贝

1、什么是浅拷贝

浅拷贝是按位拷贝对象，它会创建一个新对象，这个对象有着原始对象属性值的一份精确拷贝。如果属性是基本类型，拷贝的就是基本类型的值；如果属性是内存地址（引用类型），拷贝的就是内存地址 ，因此如果其中一个对象改变了这个地址，就会影响到另一个对象。



在上图中，SourceObject有一个int类型的属性 "field1"和一个引用类型属性"refObj"（引用ContainedObject类型的对象）。当对SourceObject做浅拷贝时，创建了CopiedObject，它有一个包含"field1"拷贝值的属性"field2"以及仍指向refObj本身的引用。由于"field1"是基本类型，所以只是将它的值拷贝给"field2"，但是由于"refObj"是一个引用类型, 所以CopiedObject指向"refObj"相同的地址。因此对SourceObject中的"refObj"所做的任何改变都会影响到CopiedObject。

2、如何实现浅拷贝

下面是实现浅拷贝的一个例子

```
public class Subject {

   private String name;

   public Subject(String s) {
      name = s;
   }

   public String getName() {
      return name;
   }

   public void setName(String s) {
      name = s;
   }
}
```

```
public class Student implements Cloneable {

    // 对象引用
    private Subject subj;

    private String name;

    public Student(String s, String sub) {
        name = s;
        subj = new Subject(sub);
    }

    public Subject getSubj() {
        return subj;
    }

    public String getName() {
        return name;
    }

    public void setName(String s) {
        name = s;
    }

    /**
     *  重写clone()方法
     * @return
     */
    public Object clone() {
        //浅拷贝
        try {
            // 直接调用父类的clone()方法
            return super.clone();
        } catch (CloneNotSupportedException e) {
            return null;
        }
    }
}
```

```
public class CopyTest {

    public static void main(String[] args) {
        // 原始对象
        Student stud = new Student("John", "Algebra");
        System.out.println("Original Object: " + stud.getName()
+ " - " + stud.getSubj().getName());

        // 拷贝对象
        Student clonedStud = (Student) stud.clone();
        System.out.println("Cloned Object: " + clonedStud.getNam
e() + " - " + clonedStud.getSubj().getName());

        // 原始对象和拷贝对象是否一样：
        System.out.println("Is Original Object the same with Clo
ned Object: " + (stud == clonedStud));
        // 原始对象和拷贝对象的name属性是否一样
        System.out.println("Is Original Object's field name the
same with Cloned Object: " + (stud.getName() == clonedStud.getNa
me()));
        // 原始对象和拷贝对象的subj属性是否一样
        System.out.println("Is Original Object's field subj the
same with Cloned Object: " + (stud.getSubj() == clonedStud.getSu
bj()));

        stud.setName("Dan");
        stud.getSubj().setName("Physics");

        System.out.println("Original Object after it is updated:
 " + stud.getName() + " - " + stud.getSubj().getName());
        System.out.println("Cloned Object after updating origina
l object: " + clonedStud.getName() + " - " + clonedStud.getSubj(
).getName());
    }
}
```

输出结果如下：

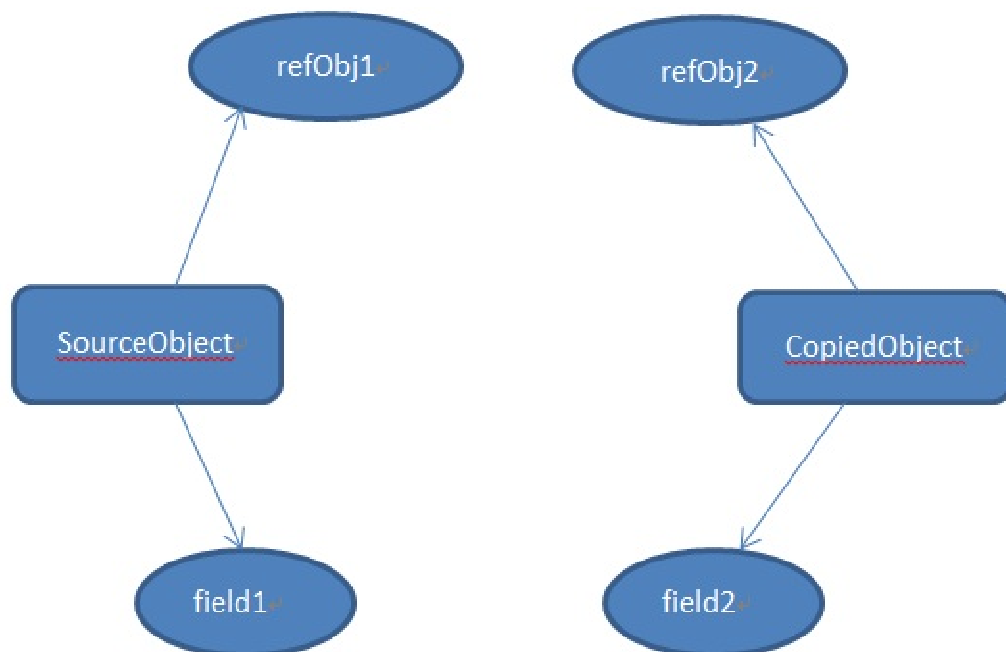
```
Original Object: John - Algebra
Cloned Object: John - Algebra
Is Original Object the same with Cloned Object: false
Is Original Object's field name the same with Cloned Object: tru
e
Is Original Object's field subj the same with Cloned Object: tru
e
Original Object after it is updated: Dan - Physics
Cloned Object after updating original object: John - Physics
```

在这个例子中，我让要拷贝的类Student实现了Clonable接口并重写Object类的clone()方法，然后在方法内部调用super.clone()方法。从输出结果中我们可以看到，对原始对象stud的"name"属性所做的改变并没有影响到拷贝对象clonedStud，但是对引用对象subj的"name"属性所做的改变影响到了拷贝对象clonedStud。

三、深拷贝

1、什么是深拷贝

深拷贝会拷贝所有的属性,并拷贝属性指向的动态分配的内存。当对象和它所引用的对象一起拷贝时即发生深拷贝。深拷贝相比于浅拷贝速度较慢并且花销较大。



在上图中，SourceObject有一个int类型的属性 "field1"和一个引用类型属性"refObj1"（引用ContainedObject类型的对象）。当对SourceObject做深拷贝时，创建了CopiedObject，它有一个包含"field1"拷贝值的属性"field2"以及包含"refObj1"拷贝值的引用类型属性"refObj2" 。因此对SourceObject中的"refObj"所做的任何改变都不会影响到CopiedObject

2、如何实现深拷贝

下面是实现深拷贝的一个例子。只是在浅拷贝的例子上做了一点小改动，Subject和CopyTest 类都没有变化。

```
public class Student implements Cloneable {
   // 对象引用
   private Subject subj;

   private String name;

   public Student(String s, String sub) {
      name = s;
      subj = new Subject(sub);
   }

   public Subject getSubj() {
      return subj;
   }

   public String getName() {
      return name;
   }

   public void setName(String s) {
      name = s;
   }

   /**
    * 重写clone()方法
    *
    * @return
    */
   public Object clone() {
      // 深拷贝，创建拷贝类的一个新对象，这样就和原始对象相互独立
      Student s = new Student(name, subj.getName());
      return s;
   }
}
```

输出结果如下：

```
Original Object: John - Algebra
Cloned Object: John - Algebra
Is Original Object the same with Cloned Object: false
Is Original Object's field name the same with Cloned Object: tru
e
Is Original Object's field subj the same with Cloned Object: fal
se
Original Object after it is updated: Dan - Physics
Cloned Object after updating original object: John - Algebra
```

很容易发现clone()方法中的一点变化。因为它是深拷贝，所以你需要创建拷贝类的一个对象。因为在Student类中有对象引用，所以需要在Student类中实现Cloneable接口并且重写clone方法。

3、通过序列化实现深拷贝

也可以通过序列化来实现深拷贝。序列化是干什么的?它将整个对象图写入到一个持久化存储文件中并且当需要的时候把它读取回来, 这意味着当你需要把它读取回来时你需要整个对象图的一个拷贝。这就是当你深拷贝一个对象时真正需要的东西。请注意，当你通过序列化进行深拷贝时，必须确保对象图中所有类都是可序列化的。

```java
public class ColoredCircle implements Serializable {

    private int x;
    private int y;

    public ColoredCircle(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return x;
    }

    public void setX(int x) {
        this.x = x;
    }

    public int getY() {
        return y;
    }

    public void setY(int y) {
        this.y = y;
    }

    @Override
    public String toString() {
        return "x=" + x + ", y=" + y;
    }
}
```

```java
public class DeepCopy {

    public static void main(String[] args) throws IOException {
        ObjectOutputStream oos = null;
        ObjectInputStream ois = null;
```

```
        try {
            // 创建原始的可序列化对象
            ColoredCircle c1 = new ColoredCircle(100, 100);
            System.out.println("Original = " + c1);

            ColoredCircle c2 = null;

            // 通过序列化实现深拷贝
            ByteArrayOutputStream bos = new ByteArrayOutputStream()
;
            oos = new ObjectOutputStream(bos);
            // 序列化以及传递这个对象
            oos.writeObject(c1);
            oos.flush();
            ByteArrayInputStream bin = new        ByteArrayInputStr
eam(bos.toByteArray());
            ois = new ObjectInputStream(bin);
            // 返回新的对象
            c2 = (ColoredCircle) ois.readObject();

            // 校验内容是否相同
            System.out.println("Copied   = " + c2);
            // 改变原始对象的内容
            c1.setX(200);
            c1.setY(200);
            // 查看每一个现在的内容
            System.out.println("Original = " + c1);
            System.out.println("Copied   = " + c2);
        } catch (Exception e) {
            System.out.println("Exception in main = " + e);
        } finally {
            oos.close();
            ois.close();
        }
    }
}
```

输出结果如下：

```
Original = x=100, y=100
Copied   = x=100, y=100
Original = x=200, y=200
Copied   = x=100, y=100
```

这里，你只需要做以下几件事儿:

- 确保对象图中的所有类都是可序列化的
- 创建输入输出流
- 使用这个输入输出流来创建对象输入和对象输出流
- 将你想要拷贝的对象传递给对象输出流
- 从对象输入流中读取新的对象并且转换回你所发送的对象的类

在这个例子中，我创建了一个ColoredCircle对象c1然后将它序列化 (将它写到ByteArrayOutputStream中). 然后我反序列化这个序列化后的对象并将它保存到c2中。随后我修改了原始对象c1。然后结果如你所见，c1不同于c2，对c1所做的任何修改都不会影响c2。

注意，序列化这种方式有其自身的限制和问题：

因为无法序列化transient变量, 使用这种方法将无法拷贝transient变量。

再就是性能问题。创建一个socket, 序列化一个对象, 通过socket传输它, 然后反序列化它，这个过程与调用已有对象的方法相比是很慢的。所以在性能上会有天壤之别。如果性能对你的代码来说是至关重要的，建议不要使用这种方式。它比通过实现Clonable接口这种方式来进行深拷贝几乎多花100倍的时间。

四、延迟拷贝

延迟拷贝是浅拷贝和深拷贝的一个组合，实际上很少会使用。 当最开始拷贝一个对象时，会使用速度较快的浅拷贝，还会使用一个计数器来记录有多少对象共享这个数据。当程序想要修改原始的对象时，它会决定数据是否被共享（通过检查计数器）并根据需要进行深拷贝。

延迟拷贝从外面看起来就是深拷贝，但是只要有可能它就会利用浅拷贝的速度。当原始对象中的引用不经常改变的时候可以使用延迟拷贝。由于存在计数器，效率下降很高，但只是常量级的开销。而且, 在某些情况下, 循环引用会导致一些问题。

五、如何选择

如果对象的属性全是基本类型的，那么可以使用浅拷贝，但是如果对象有引用属性，那就要基于具体的需求来选择浅拷贝还是深拷贝。我的意思是如果对象引用任何时候都不会被改变，那么没必要使用深拷贝，只需要使用浅拷贝就行了。如果对象引用经常改变，那么就要使用深拷贝。没有一成不变的规则，一切都取决于具体需求。

一、transient的作用及使用方法

我们都知道一个对象只要实现了Serilizable接口，这个对象就可以被序列化，java的这种序列化模式为开发者提供了很多便利，我们可以不必关系具体序列化的过程，只要这个类实现了Serilizable接口，这个类的所有属性和方法都会自动序列化。

然而在实际开发过程中，我们常常会遇到这样的问题，这个类的有些属性需要序列化，而其他属性不需要被序列化，打个比方，如果一个用户有一些敏感信息（如密码，银行卡号等），为了安全起见，不希望在网络操作（主要涉及到序列化操作，本地序列化缓存也适用）中被传输，这些信息对应的变量就可以加上transient关键字。换句话说，这个字段的生命周期仅存于调用者的内存中而不会写到磁盘里持久化。

总之，java 的transient关键字为我们提供了便利，你只需要实现Serilizable接口，将不需要序列化的属性前添加关键字transient，序列化对象的时候，这个属性就不会序列化到指定的目的地中。

示例code如下：

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;

/**
 * @description 使用transient关键字不序列化某个变量
 *          注意读取的时候，读取数据的顺序一定要和存放数据的顺序保持一致
 *
 * @author Alexia
 * @date  2013-10-15
 */
public class TransientTest {

    public static void main(String[] args) {
```

```
        User user = new User();
        user.setUsername("Alexia");
        user.setPasswd("123456");

        System.out.println("read before Serializable: ");
        System.out.println("username: " + user.getUsername());
        System.err.println("password: " + user.getPasswd());

        try {
            ObjectOutputStream os = new ObjectOutputStream(
                    new FileOutputStream("C:/user.txt"));
            os.writeObject(user); // 将User对象写进文件
            os.flush();
            os.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
        try {
            ObjectInputStream is = new ObjectInputStream(new Fil
eInputStream(
                    "C:/user.txt"));
            user = (User) is.readObject(); // 从流中读取User的数据
            is.close();

            System.out.println("\nread after Serializable: ");
            System.out.println("username: " + user.getUsername()
);
            System.err.println("password: " + user.getPasswd());

        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
```

```
}

class User implements Serializable {
    private static final long serialVersionUID = 829418001491210
3005L;

    private String username;
    private transient String passwd;

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPasswd() {
        return passwd;
    }

    public void setPasswd(String passwd) {
        this.passwd = passwd;
    }

}
```

输出为：

```
read before Serializable:
username: Alexia
password: 123456

read after Serializable:
username: Alexia
password: null
```

二、transient使用小结

1）一旦变量被transient修饰，变量将不再是对象持久化的一部分，该变量内容在序列化后无法获得访问。

2）transient关键字只能修饰变量，而不能修饰方法和类。注意，本地变量是不能被transient关键字修饰的。变量如果是用户自定义类变量，则该类需要实现Serializable接口。

3）被transient关键字修饰的变量不再能被序列化，一个静态变量不管是否被transient修饰，均不能被序列化。

第三点可能有些人很迷惑，因为发现在User类中的username字段前加上static关键字后，程序运行结果依然不变，即static类型的username也读出来为“Alexia”了，这不与第三点说的矛盾吗？实际上是这样的：第三点确实没错（一个静态变量不管是否被transient修饰，均不能被序列化），反序列化后类中static型变量username的值为当前JVM中对应static变量的值，这个值是JVM中的不是反序列化得出的，不相信？好吧，下面我来证明：

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;

/**
 * @description 使用transient关键字不序列化某个变量
 *        注意读取的时候，读取数据的顺序一定要和存放数据的顺序保持一致
 *
 * @author Alexia
 * @date  2013-10-15
 */
public class TransientTest {

    public static void main(String[] args) {

        User user = new User();
        user.setUsername("Alexia");
        user.setPasswd("123456");
```

```
        System.out.println("read before Serializable: ");
        System.out.println("username: " + user.getUsername());
        System.err.println("password: " + user.getPasswd());

        try {
            ObjectOutputStream os = new ObjectOutputStream(
                    new FileOutputStream("C:/user.txt"));
            os.writeObject(user); // 将User对象写进文件
            os.flush();
            os.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
        try {
            // 在反序列化之前改变username的值
            User.username = "jmwang";

            ObjectInputStream is = new ObjectInputStream(new Fil
eInputStream(
                    "C:/user.txt"));
            user = (User) is.readObject(); // 从流中读取User的数据
            is.close();

            System.out.println("\nread after Serializable: ");
            System.out.println("username: " + user.getUsername()
);
            System.err.println("password: " + user.getPasswd());

        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

```
class User implements Serializable {
    private static final long serialVersionUID = 829418001491210
3005L;

    public static String username;
    private transient String passwd;

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPasswd() {
        return passwd;
    }

    public void setPasswd(String passwd) {
        this.passwd = passwd;
    }

}
```

输出为：

```
read before Serializable:
username: Alexia
password: 123456

read after Serializable:
username: jmwang
password: null
```

三、transient使用细节——被transient关键字修饰的变量真的不能被序列化吗？

思考下面的例子：

```
import java.io.Externalizable;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectInputStream;
import java.io.ObjectOutput;
import java.io.ObjectOutputStream;

/**
 * @descripiton Externalizable接口的使用
 *
 * @author Alexia
 * @date 2013-10-15
 *
 */
public class ExternalizableTest implements Externalizable {

    private transient String content = "是的，我将会被序列化，不管我
是否被transient关键字修饰";

    @Override
    public void writeExternal(ObjectOutput out) throws IOExcepti
on {
        out.writeObject(content);
    }

    @Override
    public void readExternal(ObjectInput in) throws IOException,
            ClassNotFoundException {
        content = (String) in.readObject();
    }

    public static void main(String[] args) throws Exception {

        ExternalizableTest et = new ExternalizableTest();
        ObjectOutput out = new ObjectOutputStream(new FileOutput
```

```
Stream(
                new File("test")));
        out.writeObject(et);

        ObjectInput in = new ObjectInputStream(new FileInputStre
am(new File(
                "test")));
        et = (ExternalizableTest) in.readObject();
        System.out.println(et.content);

        out.close();
        in.close();
    }
}
```

content变量会被序列化吗？好吧，我把答案都输出来了，是的，运行结果就是：

```
是的，我将会被序列化，不管我是否被transient关键字修饰
```

这是为什么呢，不是说类的变量被transient关键字修饰以后将不能序列化了吗？

我们知道在Java中，对象的序列化可以通过实现两种接口来实现，若实现的是Serializable接口，则所有的序列化将会自动进行，若实现的是Externalizable接口，则没有任何东西可以自动序列化，需要在writeExternal方法中进行手工指定所要序列化的变量，这与是否被transient修饰无关。因此第二个例子输出的是变量content初始化的内容，而不是null。

网上有很多人探讨Java中异常捕获机制try...catch...finally块中的finally语句是不是一定会被执行？很多人都说不是，当然他们的回答是正确的，经过我试验，至少有两种情况下**finally**语句是不会被执行的：

（**1**）**try**语句没有被执行到，如在**try**语句之前就返回了，这样**finally**语句就不会执行，这也说明了**finally**语句被执行的必要而非充分条件是：相应的**try**语句一定被执行到。

（**2**）在**try**块中有`System.exit(0);`这样的语句，`System.exit(0);`是终止**Java**虚拟机**JVM**的，连**JVM**都停止了，所有都结束了，当然**finally**语句也不会被执行到。

当然还有很多人探讨finally语句的执行与return的关系，颇为让人迷惑，不知道finally语句是在try的return之前执行还是之后执行？我也是一头雾水，我觉得他们的说法都不正确，我觉得应该是：**finally**语句是在**try**的**return**语句执行之后，**return**返回之前执行。这样的说法有点矛盾，也许是我表述不太清楚，下面我给出自己试验的一些结果和示例进行佐证，有什么问题欢迎大家提出来。

1. finally语句在return语句执行之后return返回之前执行的。

```
public class FinallyTest1 {

    public static void main(String[] args) {

        System.out.println(test1());
    }

    public static int test1() {
        int b = 20;

        try {
            System.out.println("try block");

            return b += 80;
        }
        catch (Exception e) {

            System.out.println("catch block");
        }
        finally {

            System.out.println("finally block");

            if (b > 25) {
                System.out.println("b>25, b = " + b);
            }
        }

        return b;
    }

}
```

运行结果是：

```
try block
finally block
b>25, b = 100
100
```

说明return语句已经执行了再去执行finally语句，不过并没有直接返回，而是等finally语句执行完了再返回结果。

如果觉得这个例子还不足以说明这个情况的话，下面再加个例子加强证明结论：

```
public class FinallyTest1 {

    public static void main(String[] args) {

        System.out.println(test11());
    }

    public static String test11() {
        try {
            System.out.println("try block");

           return test12();
      } finally {
           System.out.println("finally block");
       }
  }

  public static String test12() {
       System.out.println("return statement");

       return "after return";
   }

}
```

运行结果为：

```
try block
return statement
finally block
after return
```

说明try中的return语句先执行了但并没有立即返回，等到finally执行结束后再返回。

这里大家可能会想：如果finally里也有return语句，那么是不是就直接返回了，try中的return就不能返回了？看下面。

2. finally块中的return语句会覆盖try块中的return返回。

```
public class FinallyTest2 {

    public static void main(String[] args) {

        System.out.println(test2());
    }

    public static int test2() {
        int b = 20;

        try {
            System.out.println("try block");

            return b += 80;
        } catch (Exception e) {

            System.out.println("catch block");
        } finally {

            System.out.println("finally block");

            if (b > 25) {
                System.out.println("b>25, b = " + b);
            }

            return 200;
        }

        // return b;
    }

}
```

运行结果是：

```
try block
finally block
b>25, b = 100
200
```

这说明finally里的return直接返回了，就不管try中是否还有返回语句，这里还有个小细节需要注意，finally里加上return过后，finally外面的return b就变成不可到达语句了，也就是永远不能被执行到，所以需要注释掉否则编译器报错。

这里大家可能又想：如果finally里没有return语句，但修改了b的值，那么try中return返回的是修改后的值还是原值？看下面。

3. 如果finally语句中没有return语句覆盖返回值，那么原来的返回值可能因为finally里的修改而改变也可能不变。

测试用例**1**：

```
public class FinallyTest3 {

    public static void main(String[] args) {

        System.out.println(test3());
    }

    public static int test3() {
        int b = 20;

        try {
            System.out.println("try block");

            return b += 80;
        } catch (Exception e) {

            System.out.println("catch block");
        } finally {

            System.out.println("finally block");

            if (b > 25) {
                System.out.println("b>25, b = " + b);
            }

            b = 150;
        }

        return 2000;
    }

}
```

运行结果是：

```
try block
finally block
b>25, b = 100
100
```

测试用例**2**：

```
import java.util.*;

public class FinallyTest6
{
    public static void main(String[] args) {
        System.out.println(getMap().get("KEY").toString());
    }

    public static Map<String, String> getMap() {
        Map<String, String> map = new HashMap<String, String>();
        map.put("KEY", "INIT");

        try {
            map.put("KEY", "TRY");
            return map;
        }
        catch (Exception e) {
            map.put("KEY", "CATCH");
        }
        finally {
            map.put("KEY", "FINALLY");
            map = null;
        }

        return map;
    }
}
```

运行结果是：

```
FINALLY
```

为什么测试用例1中finally里的b = 150;并没有起到作用而测试用例2中finally的map.put("KEY", "FINALLY");起了作用而map = null;却没起作用呢？这就是Java到底是传值还是传址的问题了，具体请看精选30道Java笔试题解答，里面有详细的解答，简单来说就是：Java中只有传值没有传址，这也是为什么map = null这句不起作用。这同时也说明了返回语句是try中的return语句而不是 finally外面的return b;这句，不相信的话可以试下，将return b;改为return 294，对原来的结果没有一点影响。

这里大家可能又要想：是不是每次返回的一定是try中的return语句呢？那么finally外的return b不是一点作用没吗？请看下面。

4. try块里的return语句在异常的情况下不会被执行，这样具体返回哪个看情况。

```
public class FinallyTest4 {

    public static void main(String[] args) {

        System.out.println(test4());
    }

    public static int test4() {
        int b = 20;

        try {
            System.out.println("try block");

            b = b / 0;

            return b += 80;
        } catch (Exception e) {

            b += 15;
            System.out.println("catch block");
        } finally {

            System.out.println("finally block");

            if (b > 25) {
                System.out.println("b>25, b = " + b);
            }

            b += 50;
        }

        return b;
    }

}
```

运行结果是：

```
try block
catch block
finally block
b>25, b = 35
85
```

这里因 为在return之前发生了除0异常，所以try中的return不会被执行到，而是接着执行捕获异常的catch 语句和最终的finally语句，此时两者对b的修改都影响了最终的返回值，这时return b;就起到作用了。当然如果你这里将return b改为return 300什么的，最后返回的就是300，这毋庸置疑。

这里大家可能又有疑问：如果catch中有return语句呢？当然只有在异常的情况下才有可能会执行，那么是在finally之前就返回吗？看下面。

5. 当发生异常后，catch中的return执行情况与未发生异常时try中return的执行情况完全一样。

```
public class FinallyTest5 {

    public static void main(String[] args) {

        System.out.println(test5());
    }

    public static int test5() {
        int b = 20;

        try {
            System.out.println("try block");

            b = b /0;

            return b += 80;
        } catch (Exception e) {

            System.out.println("catch block");
            return b += 15;
        } finally {

            System.out.println("finally block");

            if (b > 25) {
                System.out.println("b>25, b = " + b);
            }

            b += 50;
        }

        //return b;
    }

}
```

运行结果如下：

```
try block
catch block
finally block
b>25, b = 35
35
```

说明了发生异常后，catch中的return语句先执行，确定了返回值后再去执行finally块，执行完了catch再返回，finally里对b的改变对返回值无影响，原因同前面一样，也就是说情况与try中的return语句执行完全一样。

最后总结：**finally**块的语句在**try**或**catch**中的**return**语句执行之后返回之前执行且**finally**里的修改语句可能影响也可能不影响**try**或**catch**中 **return**已经确定的返回值，若**finally**里也有**return**语句则覆盖**try**或**catch**中的**return**语句直接返回。

1. 简介

毫无疑问，Java 8是Java自Java 5（发布于2004年）之后的最重要的版本。这个版本包含语言、编译器、库、工具和JVM等方面的十多个新特性。在本文中我们将学习这些新特性，并用实际的例子说明在什么场景下适合使用。

这个教程包含Java开发者经常面对的几类问题：

- 语言
- 编译器
- 库
- 工具
- 运行时（JVM）

2. Java语言的新特性

Java 8是Java的一个重大版本，有人认为，虽然这些新特性领Java开发人员十分期待，但同时也需要花不少精力去学习。在这一小节中，我们将介绍Java 8的大部分新特性。

2.1 Lambda表达式和函数式接口

Lambda表达式（也称为闭包）是Java 8中最大和最令人期待的语言改变。它允许我们将函数当成参数传递给某个方法，或者把代码本身当作数据处理：函数式开发者非常熟悉这些概念。很多JVM平台上的语言（Groovy、Scala等）从诞生之日就支持Lambda表达式，但是Java开发者没有选择，只能使用匿名内部类代替Lambda表达式。

Lambda的设计耗费了很多时间和很大的社区力量，最终找到一种折中的实现方案，可以实现简洁而紧凑的语言结构。最简单的Lambda表达式可由逗号分隔的参数列表、**->**符号和语句块组成，例如：

```
Arrays.asList( "a", "b", "d" ).forEach( e -> System.out.println(
 e ) );
```

在上面这个代码中的参数**e**的类型是由编译器推理得出的，你也可以显式指定该参数的类型，例如：

```
Arrays.asList( "a", "b", "d" ).forEach( ( String e ) -> System.o
ut.println( e ) );
```

如果Lambda表达式需要更复杂的语句块，则可以使用花括号将该语句块括起来，类似于Java中的函数体，例如：

```
Arrays.asList( "a", "b", "d" ).forEach( e -> {
    System.out.print( e );
    System.out.print( e );
} );
```

Lambda表达式可以引用类成员和局部变量（会将这些变量隐式得转换成**final**的），例如下列两个代码块的效果完全相同：

```
String separator = ",";
Arrays.asList( "a", "b", "d" ).forEach(
    ( String e ) -> System.out.print( e + separator ) );
```

和

```
final String separator = ",";
Arrays.asList( "a", "b", "d" ).forEach(
    ( String e ) -> System.out.print( e + separator ) );
```

Lambda表达式有返回值，返回值的类型也由编译器推理得出。如果Lambda表达式中的语句块只有一行，则可以不用使用**return**语句，下列两个代码片段效果相同：

```
Arrays.asList( "a", "b", "d" ).sort( ( e1, e2 ) -> e1.compareTo(
 e2 ) );
```

和

```
Arrays.asList( "a", "b", "d" ).sort( ( e1, e2 ) -> {
    int result = e1.compareTo( e2 );
    return result;
} );
```

Lambda的设计者们为了让现有的功能与Lambda表达式良好兼容，考虑了很多方法，于是产生了函数接口这个概念。函数接口指的是只有一个函数的接口，这样的接口可以隐式转换为Lambda表达式。**java.lang.Runnable**和**java.util.concurrent.Callable**是函数式接口的最佳例子。在实践中，函数式接口非常脆弱：只要某个开发者在该接口中添加一个函数，则该接口就不再是函数式接口进而导致编译失败。为了克服这种代码层面的脆弱性，并显式说明某个接口是函数式接口，Java 8 提供了一个特殊的注解**@FunctionalInterface**（Java 库中的所有相关接口都已经带有这个注解了），举个简单的函数式接口的定义：

```
@FunctionalInterface
public interface Functional {
    void method();
}
```

不过有一点需要注意，默认方法和静态方法不会破坏函数式接口的定义，因此如下的代码是合法的。

```
@FunctionalInterface
public interface FunctionalDefaultMethods {
    void method();

    default void defaultMethod() {
    }
}
```

Lambda表达式作为Java 8的最大卖点，它有潜力吸引更多的开发者加入到JVM平台，并在纯Java编程中使用函数式编程的概念。如果你需要了解更多Lambda表达式的细节，可以参考官方文档。

2.2 接口的默认方法和静态方法

Java 8使用两个新概念扩展了接口的含义：默认方法和静态方法。默认方法使得接口有点类似traits，不过要实现的目标不一样。默认方法使得开发者可以在 不破坏二进制兼容性的前提下，往现存接口中添加新的方法，即不强制那些实现了该接口的类也同时实现这个新加的方法。

默认方法和抽象方法之间的区别在于抽象方法需要实现，而默认方法不需要。接口提供的默认方法会被接口的实现类继承或者覆写，例子代码如下：

```
private interface Defaulable {
    // Interfaces now allow default methods, the implementer may
 or
    // may not implement (override) them.
    default String notRequired() {
        return "Default implementation";
    }
}

private static class DefaultableImpl implements Defaulable {
}

private static class OverridableImpl implements Defaulable {
    @Override
    public String notRequired() {
        return "Overridden implementation";
    }
}
```

Defaulable接口使用关键字**default**定义了一个默认方法**notRequired()**。**DefaultableImpl**类实现了这个接口，同时默认继承了这个接口中的默认方法；**OverridableImpl**类也实现了这个接口，但覆写了该接口的默认方法，并提供了一个不同的实现。

Java 8带来的另一个有趣的特性是在接口中可以定义静态方法，例子代码如下：

```
private interface DefaulableFactory {
    // Interfaces now allow static methods
    static Defaulable create( Supplier< Defaulable > supplier )
{
        return supplier.get();
    }
}
```

下面的代码片段整合了默认方法和静态方法的使用场景：

```
public static void main( String[] args ) {
    Defaulable defaulable = DefaulableFactory.create( Defaultabl
eImpl::new );
    System.out.println( defaulable.notRequired() );

    defaulable = DefaulableFactory.create( OverridableImpl::new
);
    System.out.println( defaulable.notRequired() );
}
```

这段代码的输出结果如下：

```
Default implementation
Overridden implementation
```

由于JVM上的默认方法的实现在字节码层面提供了支持，因此效率非常高。默认方法允许在不打破现有继承体系的基础上改进接口。该特性在官方库中的应用是：给**java.util.Collection**接口添加新方法，如**stream()**、**parallelStream()**、**forEach()**和**removeIf()**等等。

尽管默认方法有这么多好处，但在实际开发中应该谨慎使用：在复杂的继承体系中，默认方法可能引起歧义和编译错误。如果你想了解更多细节，可以参考官方文档。

2.3 方法引用

方法引用使得开发者可以直接引用现存的方法、Java类的构造方法或者实例对象。方法引用和Lambda表达式配合使用，使得java类的构造方法看起来紧凑而简洁，没有很多复杂的模板代码。

西门的例子中，**Car**类是不同方法引用的例子，可以帮助读者区分四种类型的方法引用。

```
public static class Car {
    public static Car create( final Supplier< Car > supplier ) {
        return supplier.get();
    }

    public static void collide( final Car car ) {
        System.out.println( "Collided " + car.toString() );
    }

    public void follow( final Car another ) {
        System.out.println( "Following the " + another.toString(
) );
    }

    public void repair() {
        System.out.println( "Repaired " + this.toString() );
    }
}
```

第一种方法引用的类型是构造器引用，语法是**Class::new**，或者更一般的形式：**Class::new**。注意：这个构造器没有参数。

```
final Car car = Car.create( Car::new );
final List< Car > cars = Arrays.asList( car );
```

第二种方法引用的类型是静态方法引用，语法是**Class::static_method**。注意：这个方法接受一个Car类型的参数。

```
cars.forEach( Car::collide );
```

第三种方法引用的类型是某个类的成员方法的引用，语法是**Class::method**，注意，这个方法没有定义入参：

```
cars.forEach( Car::repair );
```

第四种方法引用的类型是某个实例对象的成员方法的引用，语法是**instance::method**。注意：这个方法接受一个Car类型的参数：

```
final Car police = Car.create( Car::new );
cars.forEach( police::follow );
```

运行上述例子，可以在控制台看到如下输出（Car实例可能不同）：

```
Collided com.javacodegeeks.java8.method.references.MethodReferen
ces$Car@7a81197d
Repaired com.javacodegeeks.java8.method.references.MethodReferen
ces$Car@7a81197d
Following the com.javacodegeeks.java8.method.references.MethodRe
ferences$Car@7a81197d
```

如果想了解和学习更详细的内容，可以参考官方文档

2.4 重复注解

自从Java 5中引入注解以来，这个特性开始变得非常流行，并在各个框架和项目中被广泛使用。不过，注解有一个很大的限制是：在同一个地方不能多次使用同一个注解。Java 8打破了这个限制，引入了重复注解的概念，允许在同一个地方多次使用同一个注解。

在Java 8中使用**@Repeatable**注解定义重复注解，实际上，这并不是语言层面的改进，而是编译器做的一个trick，底层的技术仍然相同。可以利用下面的代码说明：

```
package com.javacodegeeks.java8.repeatable.annotations;

import java.lang.annotation.ElementType;
import java.lang.annotation.Repeatable;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

public class RepeatingAnnotations {
    @Target( ElementType.TYPE )
    @Retention( RetentionPolicy.RUNTIME )
    public @interface Filters {
        Filter[] value();
    }

    @Target( ElementType.TYPE )
    @Retention( RetentionPolicy.RUNTIME )
    @Repeatable( Filters.class )
    public @interface Filter {
        String value();
    };

    @Filter( "filter1" )
    @Filter( "filter2" )
    public interface Filterable {
    }

    public static void main(String[] args) {
        for( Filter filter: Filterable.class.getAnnotationsByTyp
e( Filter.class ) ) {
            System.out.println( filter.value() );
        }
    }
}
```

正如我们所见，这里的**Filter**类使用@Repeatable(Filters.class)注解修饰，而**Filters**是存放**Filter**注解的容器，编译器尽量对开发者屏蔽这些细节。这样，**Filterable**接口可以用两个**Filter**注解注释（这里并没有提到任何关于Filters的信息）。

另外，反射API提供了一个新的方法：**getAnnotationsByType()**，可以返回某个类型的重复注解，例如 `Filterable.class.getAnnoation(Filters.class)` 将返回两个Filter实例，输出到控制台的内容如下所示：

```
filter1
filter2
```

如果你希望了解更多内容，可以参考官方文档。

2.5 更好的类型推断

Java 8编译器在类型推断方面有很大的提升，在很多场景下编译器可以推导出某个参数的数据类型，从而使得代码更为简洁。例子代码如下：

```
package com.javacodegeeks.java8.type.inference;

public class Value< T > {
    public static< T > T defaultValue() {
        return null;
    }

    public T getOrDefault( T value, T defaultValue ) {
        return ( value != null ) ? value : defaultValue;
    }
}
```

下列代码是**Value**类型的应用：

```
package com.javacodegeeks.java8.type.inference;

public class TypeInference {
    public static void main(String[] args) {
        final Value< String > value = new Value<>();
        value.getOrDefault( "22", Value.defaultValue() );
    }
}
```

参数**Value.defaultValue()**的类型由编译器推导得出，不需要显式指明。在Java 7中这段代码会有编译错误，除非使用 `Value.<String>defaultValue()` 。

2.6 拓宽注解的应用场景

Java 8拓宽了注解的应用场景。现在，注解几乎可以使用在任何元素上：局部变量、接口类型、超类和接口实现类，甚至可以用在函数的异常定义上。下面是一些例子：

```
package com.javacodegeeks.java8.annotations;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import java.util.ArrayList;
import java.util.Collection;

public class Annotations {
    @Retention( RetentionPolicy.RUNTIME )
    @Target( { ElementType.TYPE_USE, ElementType.TYPE_PARAMETER
} )
    public @interface NonEmpty {
    }

    public static class Holder< @NonEmpty T > extends @NonEmpty
Object {
        public void method() throws @NonEmpty Exception {

        }
    }

    @SuppressWarnings( "unused" )
    public static void main(String[] args) {
        final Holder< String > holder = new @NonEmpty Holder< St
ring >();
        @NonEmpty Collection< @NonEmpty String > strings = new A
rrayList<>();
    }
}
```

ElementType.TYPE_USER和**ElementType.TYPE_PARAMETER**是Java 8新增的两个注解，用于描述注解的使用场景。Java 语言也做了对应的改变，以识别这些新增的注解。

3. Java编译器的新特性

3.1 参数名称

为了在运行时获得Java程序中方法的参数名称，老一辈的Java程序员必须使用不同方法，例如Paranamer liberary。Java 8终于将这个特性规范化，在语言层面（使用反射API和**Parameter.getName()**方法）和字节码层面（使用新的**javac**编译器以及**-parameters**参数）提供支持。

```
package com.javacodegeeks.java8.parameter.names;

import java.lang.reflect.Method;
import java.lang.reflect.Parameter;

public class ParameterNames {
    public static void main(String[] args) throws Exception {
        Method method = ParameterNames.class.getMethod( "main",
String[].class );
        for( final Parameter parameter: method.getParameters() )
 {
            System.out.println( "Parameter: " + parameter.getNam
e() );
        }
    }
}
```

在Java 8中这个特性是默认关闭的，因此如果不带**-parameters**参数编译上述代码并运行，则会输出如下结果：

```
Parameter: arg0
```

如果带**-parameters**参数，则会输出如下结果（正确的结果）：

```
Parameter: args
```

如果你使用Maven进行项目管理，则可以在**maven-compiler-plugin**编译器的配置项中配置**-parameters**参数：

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.1</version>
    <configuration>
        <compilerArgument>-parameters</compilerArgument>
        <source>1.8</source>
        <target>1.8</target>
    </configuration>
</plugin>
```

4. Java官方库的新特性

Java 8增加了很多新的工具类（date/time类），并扩展了现存的工具类，以支持现代的并发编程、函数式编程等。

4.1 Optional

Java应用中最常见的bug就是空值异常。在Java 8之前，Google Guava引入了**Optionals**类来解决**NullPointerException**，从而避免源码被各种**null**检查污染，以便开发者写出更加整洁的代码。Java 8也将**Optional**加入了官方库。

Optional仅仅是一个容易：存放T类型的值或者null。它提供了一些有用的接口来避免显式的null检查，可以参考Java 8官方文档了解更多细节。

接下来看一点使用**Optional**的例子：可能为空的值或者某个类型的值：

```
Optional< String > fullName = Optional.ofNullable( null );
System.out.println( "Full Name is set? " + fullName.isPresent()
);
System.out.println( "Full Name: " + fullName.orElseGet( () -> "[
none]" ) );
System.out.println( fullName.map( s -> "Hey " + s + "!" ).orElse
( "Hey Stranger!" ) );
```

如果**Optional**实例持有一个非空值，则**isPresent()**方法返回true，否则返回false；**orElseGet()**方法，**Optional**实例持有null，则可以接受一个lambda表达式生成的默认值；**map()**方法可以将现有的**Opetional**实例的值转换成新的值；**orElse()**方法与**orElseGet()**方法类似，但是在持有null的时候返回传入的默认值。

上述代码的输出结果如下：

```
Full Name is set? false
Full Name: [none]
Hey Stranger!
```

再看下另一个简单的例子：

```
Optional< String > firstName = Optional.of( "Tom" );
System.out.println( "First Name is set? " + firstName.isPresent(
) );
System.out.println( "First Name: " + firstName.orElseGet( () ->
"[none]" ) );
System.out.println( firstName.map( s -> "Hey " + s + "!" ).orEls
e( "Hey Stranger!" ) );
System.out.println();
```

这个例子的输出是：

```
First Name is set? true
First Name: Tom
Hey Tom!
```

如果想了解更多的细节，请参考官方文档。

4.2 Streams

新增的Stream API（java.util.stream）将生成环境的函数式编程引入了Java库中。这是目前为止最大的一次对Java库的完善，以便开发者能够写出更加有效、更加简洁和紧凑的代码。

Stream API极大得简化了集合操作（后面我们会看到不止是集合），首先看下这个叫Task的类：

```
public class Streams  {
    private enum Status {
        OPEN, CLOSED
    };

    private static final class Task {
        private final Status status;
        private final Integer points;

        Task( final Status status, final Integer points ) {
            this.status = status;
            this.points = points;
        }

        public Integer getPoints() {
            return points;
        }

        public Status getStatus() {
            return status;
        }

        @Override
        public String toString() {
            return String.format( "[%s, %d]", status, points );
        }
    }
}
```

Task类有一个分数（或伪复杂度）的概念，另外还有两种状态：OPEN或者CLOSED。现在假设有一个task集合：

```
final Collection< Task > tasks = Arrays.asList(
    new Task( Status.OPEN, 5 ),
    new Task( Status.OPEN, 13 ),
    new Task( Status.CLOSED, 8 )
);
```

首先看一个问题：在这个task集合中一共有多少个OPEN状态的点？在Java 8之前，要解决这个问题，则需要使用**foreach**循环遍历task集合；但是在Java 8中可以利用steams解决：包括一系列元素的列表，并且支持顺序和并行处理。

```
// Calculate total points of all active tasks using sum()
final long totalPointsOfOpenTasks = tasks
    .stream()
    .filter( task -> task.getStatus() == Status.OPEN )
    .mapToInt( Task::getPoints )
    .sum();

System.out.println( "Total points: " + totalPointsOfOpenTasks );
```

运行这个方法的控制台输出是：

```
Total points: 18
```

这里有很多知识点值得说。首先，tasks集合被转换成steam表示；其次，在steam上的**filter**操作会过滤掉所有CLOSED的task；第三，**mapToInt**操作基于每个task实例的**Task::getPoints**方法将task流转换成Integer集合；最后，通过**sum**方法计算总和，得出最后的结果。

在学习下一个例子之前，还需要记住一些steams（点此更多细节）的知识点。Steam之上的操作可分为中间操作和晚期操作。

中间操作会返回一个新的steam——执行一个中间操作（例如**filter**）并不会执行实际的过滤操作，而是创建一个新的steam，并将原steam中符合条件的元素放入新创建的steam。

晚期操作（例如**forEach**或者**sum**），会遍历steam并得出结果或者附带结果；在执行晚期操作之后，steam处理线已经处理完毕，就不能使用了。在几乎所有情况下，晚期操作都是立刻对steam进行遍历。

steam的另一个价值是创造性地支持并行处理（parallel processing）。对于上述的tasks集合，我们可以用下面的代码计算所有任务的点数之和：

```
// Calculate total points of all tasks
final double totalPoints = tasks
   .stream()
   .parallel()
   .map( task -> task.getPoints() ) // or map( Task::getPoints )

   .reduce( 0, Integer::sum );

System.out.println( "Total points (all tasks): " + totalPoints )
;
```

这里我们使用**parallel**方法并行处理所有的task，并使用**reduce**方法计算最终的结果。控制台输出如下：

```
Total points（all tasks）: 26.0
```

对于一个集合，经常需要根据某些条件对其中的元素分组。利用steam提供的API可以很快完成这类任务，代码如下：

```
// Group tasks by their status
final Map< Status, List< Task > > map = tasks
    .stream()
    .collect( Collectors.groupingBy( Task::getStatus ) );
System.out.println( map );
```

控制台的输出如下：

```
{CLOSED=[[CLOSED, 8]], OPEN=[[OPEN, 5], [OPEN, 13]]}
```

最后一个关于tasks集合的例子问题是：如何计算集合中每个任务的点数在集合中所占的比重，具体处理的代码如下：

```
// Calculate the weight of each tasks (as percent of total point
s)
final Collection< String > result = tasks
    .stream()                                        // Stream<
String >
    .mapToInt( Task::getPoints )                     // IntStream

    .asLongStream()                                  // LongStre
am
    .mapToDouble( points -> points / totalPoints )   // DoubleSt
ream
    .boxed()                                         // Stream<
Double >
    .mapToLong( weigth -> ( long )( weigth * 100 ) ) // LongStre
am
    .mapToObj( percentage -> percentage + "%" )      // Stream<
String>
    .collect( Collectors.toList() );                 // List< St
ring >

System.out.println( result );
```

控制台输出结果如下：

```
[19%, 50%, 30%]
```

最后，正如之前所说，Steam API不仅可以作用于Java集合，传统的IO操作（从文件或者网络一行一行得读取数据）可以受益于steam处理，这里有一个小例子：

```
final Path path = new File( filename ).toPath();
try( Stream< String > lines = Files.lines( path, StandardCharset
s.UTF_8 ) ) {
    lines.onClose( () -> System.out.println("Done!") ).forEach(
System.out::println );
}
```

Stream的方法**onClose** 返回一个等价的有额外句柄的Stream，当Stream的close（）方法被调用的时候这个句柄会被执行。Stream API、Lambda表达式还有接口默认方法和静态方法支持的方法引用，是Java 8对软件开发的现代范式的响应。

4.3 Date/Time API(JSR 310)

Java 8引入了新的Date-Time API(JSR 310)来改进时间、日期的处理。时间和日期的管理一直是最令Java开发者痛苦的问题。**java.util.Date**和后来的**java.util.Calendar**一直没有解决这个问题（甚至令开发者更加迷茫）。

因为上面这些原因，诞生了第三方库Joda-Time，可以替代Java的时间管理API。Java 8中新的时间和日期管理API深受Joda-Time影响，并吸收了很多Joda-Time的精华。新的java.time包包含了所有关于日期、时间、时区、Instant（跟日期类似但是精确到纳秒）、duration（持续时间）和时钟操作的类。新设计的API认真考虑了这些类的不变性（从java.util.Calendar吸取的教训），如果某个实例需要修改，则返回一个新的对象。

我们接下来看看java.time包中的关键类和各自的使用例子。首先，**Clock**类使用时区来返回当前的纳秒时间和日期。**Clock**可以替代**System.currentTimeMillis()**和**TimeZone.getDefault()**。

```
// Get the system clock as UTC offset
final Clock clock = Clock.systemUTC();
System.out.println( clock.instant() );
System.out.println( clock.millis() );
```

这个例子的输出结果是：

```
2014-04-12T15:19:29.282Z
1397315969360
```

第二，关注下**LocalDate**和**LocalTime**类。**LocalDate**仅仅包含ISO-8601日历系统中的日期部分；**LocalTime**则仅仅包含该日历系统中的时间部分。这两个类的对象都可以使用Clock对象构建得到。

```
// Get the local date and local time
final LocalDate date = LocalDate.now();
final LocalDate dateFromClock = LocalDate.now( clock );

System.out.println( date );
System.out.println( dateFromClock );

// Get the local date and local time
final LocalTime time = LocalTime.now();
final LocalTime timeFromClock = LocalTime.now( clock );

System.out.println( time );
System.out.println( timeFromClock );
```

上述例子的输出结果如下：

```
2014-04-12
2014-04-12
11:25:54.568
15:25:54.568
```

LocalDateTime类包含了LocalDate和LocalTime的信息，但是不包含ISO-8601日历系统中的时区信息。这里有一些关于LocalDate和LocalTime的例子：

```
// Get the local date/time
final LocalDateTime datetime = LocalDateTime.now();
final LocalDateTime datetimeFromClock = LocalDateTime.now( clock
 );

System.out.println( datetime );
System.out.println( datetimeFromClock );
```

上述这个例子的输出结果如下：

```
2014-04-12T11:37:52.309
2014-04-12T15:37:52.309
```

如果你需要特定时区的data/time信息，则可以使用**ZoneDateTime**，它保存有ISO-8601日期系统的日期和时间，而且有时区信息。下面是一些使用不同时区的例子：

```
// Get the zoned date/time
final ZonedDateTime zonedDatetime = ZonedDateTime.now();
final ZonedDateTime zonedDatetimeFromClock = ZonedDateTime.now(
clock );
final ZonedDateTime zonedDatetimeFromZone = ZonedDateTime.now( Z
oneId.of( "America/Los_Angeles" ) );

System.out.println( zonedDatetime );
System.out.println( zonedDatetimeFromClock );
System.out.println( zonedDatetimeFromZone );
```

这个例子的输出结果是：

```
2014-04-12T11:47:01.017-04:00[America/New_York]
2014-04-12T15:47:01.017Z
2014-04-12T08:47:01.017-07:00[America/Los_Angeles]
```

最后看下**Duration**类，它持有的时间精确到秒和纳秒。这使得我们可以很容易得计算两个日期之间的不同，例子代码如下：

```
// Get duration between two dates
final LocalDateTime from = LocalDateTime.of( 2014, Month.APRIL,
16, 0, 0, 0 );
final LocalDateTime to = LocalDateTime.of( 2015, Month.APRIL, 16
, 23, 59, 59 );

final Duration duration = Duration.between( from, to );
System.out.println( "Duration in days: " + duration.toDays() );
System.out.println( "Duration in hours: " + duration.toHours() )
;
```

这个例子用于计算2014年4月16日和2015年4月16日之间的天数和小时数，输出结果如下：

```
Duration in days: 365
Duration in hours: 8783
```

对于Java 8的新日期时间的总体印象还是比较积极的，一部分是因为Joda-Time的积极影响，另一部分是因为官方终于听取了开发人员的需求。如果希望了解更多细节，可以参考官方文档。

4.4 Nashorn JavaScript引擎

Java 8提供了新的Nashorn JavaScript引擎，使得我们可以在JVM上开发和运行JS应用。Nashorn JavaScript引擎是javax.script.ScriptEngine的另一个实现版本，这类Script引擎遵循相同的规则，允许Java和JavaScript交互使用，例子代码如下：

```
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName( "JavaScript" );

System.out.println( engine.getClass().getName() );
System.out.println( "Result:" + engine.eval( "function f() { ret
urn 1; }; f() + 1;" ) );
```

这个代码的输出结果如下：

```
jdk.nashorn.api.scripting.NashornScriptEngine
Result: 2
```

4.5 Base64

对Base64编码的支持已经被加入到Java 8官方库中，这样不需要使用第三方库就可以进行Base64编码，例子代码如下：

```
package com.javacodegeeks.java8.base64;

import java.nio.charset.StandardCharsets;
import java.util.Base64;

public class Base64s {
    public static void main(String[] args) {
        final String text = "Base64 finally in Java 8!";

        final String encoded = Base64
            .getEncoder()
            .encodeToString( text.getBytes( StandardCharsets.UTF
_8 ) );
        System.out.println( encoded );

        final String decoded = new String(
            Base64.getDecoder().decode( encoded ),
            StandardCharsets.UTF_8 );
        System.out.println( decoded );
    }
}
```

这个例子的输出结果如下：

```
QmFzZTY0IGZpbmFsbHkgaW4gSmF2YSA4IQ==
Base64 finally in Java 8!
```

新的Base64API也支持URL和MINE的编码解码。(**Base64.getUrlEncoder()** / **Base64.getUrlDecoder()**, **Base64.getMimeEncoder()** / **Base64.getMimeDecoder()**)。

4.6 并行数组

Java8版本新增了很多新的方法，用于支持并行数组处理。最重要的方法是**parallelSort()**，可以显著加快多核机器上的数组排序。下面的例子论证了**parallexXxx**系列的方法：

```
package com.javacodegeeks.java8.parallel.arrays;

import java.util.Arrays;
import java.util.concurrent.ThreadLocalRandom;

public class ParallelArrays {
    public static void main( String[] args ) {
        long[] arrayOfLong = new long [ 20000 ];

        Arrays.parallelSetAll( arrayOfLong,
            index -> ThreadLocalRandom.current().nextInt( 1000000
 ) );
        Arrays.stream( arrayOfLong ).limit( 10 ).forEach(
            i -> System.out.print( i + " " ) );
        System.out.println();

        Arrays.parallelSort( arrayOfLong );
        Arrays.stream( arrayOfLong ).limit( 10 ).forEach(
            i -> System.out.print( i + " " ) );
        System.out.println();
    }
}
```

上述这些代码使用**parallelSetAll()**方法生成20000个随机数，然后使用**parallelSort()**方法进行排序。这个程序会输出乱序数组和排序数组的前10个元素。上述例子的代码输出的结果是：

```
Unsorted: 591217 891976 443951 424479 766825 351964 242997 64283
9 119108 552378
Sorted: 39 220 263 268 325 607 655 678 723 793
```

4.7 并发性

基于新增的lambda表达式和steam特性，为Java 8中为**java.util.concurrent.ConcurrentHashMap**类添加了新的方法来支持聚焦操作；另外，也为**java.util.concurrentForkJoinPool**类添加了新的方法来支持通用线程池操作（更多内容可以参考我们的并发编程课程）。

Java 8还添加了新的**java.util.concurrent.locks.StampedLock**类，用于支持基于容量的锁——该锁有三个模型用于支持读写操作（可以把这个锁当做是**java.util.concurrent.locks.ReadWriteLock**的替代者）。

在**java.util.concurrent.atomic**包中也新增了不少工具类，列举如下：

- DoubleAccumulator
- DoubleAdder
- LongAccumulator
- LongAdder

5. 新的Java工具

Java 8提供了一些新的命令行工具，这部分会讲解一些对开发者最有用的工具。

5.1 Nashorn引擎：jjs

jjs是一个基于标准Nashorn引擎的命令行工具，可以接受js源码并执行。例如，我们写一个**func.js**文件，内容如下：

```
function f() {
     return 1;
};

print( f() + 1 );
```

可以在命令行中执行这个命令：`jjs func.js`，控制台输出结果是：

```
2
```

如果需要了解细节，可以参考官方文档。

5.2 类依赖分析器：jdeps

jdeps是一个相当棒的命令行工具，它可以展示包层级和类层级的Java类依赖关系，它以**.class**文件、目录或者Jar文件为输入，然后会把依赖关系输出到控制台。

我们可以利用jedps分析下Spring Framework库，为了让结果少一点，仅仅分析一个JAR文件：**org.springframework.core-3.0.5.RELEASE.jar**。

```
jdeps org.springframework.core-3.0.5.RELEASE.jar
```

这个命令会输出很多结果，我们仅看下其中的一部分：依赖关系按照包分组，如果在classpath上找不到依赖，则显示"not found".

```
org.springframework.core-3.0.5.RELEASE.jar -> C:\Program Files\J
ava\jdk1.8.0\jre\lib\rt.jar
   org.springframework.core (org.springframework.core-3.0.5.RELE
ASE.jar)
      -> java.io
      -> java.lang
      -> java.lang.annotation
      -> java.lang.ref
      -> java.lang.reflect
      -> java.util
      -> java.util.concurrent
      -> org.apache.commons.logging                         not
found
      -> org.springframework.asm                            not
found
      -> org.springframework.asm.commons                    not
found
   org.springframework.core.annotation (org.springframework.core
-3.0.5.RELEASE.jar)
      -> java.lang
      -> java.lang.annotation
      -> java.lang.reflect
      -> java.util
```

更多的细节可以参考官方文档。

6. JVM的新特性

使用**Metaspace**（JEP 122）代替持久代（**PermGen** space）。在JVM参数方面，使用**-XX:MetaSpaceSize**和**-XX:MaxMetaspaceSize**代替原来的**-XX:PermSize**和**-XX:MaxPermSize**。

一、前言

本部分内容是关于Java并发的一些知识总结，既是学习的难点，同时也是面试中几乎必问的知识点。

面试中可能会问的一些问题：

- 创建线程的方式
- Synchronized/ReentrantLock
- 生产者/消费者模式
- volatile关键字
- 乐观锁/悲观锁
- 死锁
- 了解的并发集合

因此针对以上问题，整理了相关内容。

二、目录

一、继承Thread类创建线程类

（1）定义Thread类的子类，并重写该类的run方法，该run方法的方法体就代表了线程要完成的任务。因此把run()方法称为执行体。

（2）创建Thread子类的实例，即创建了线程对象。

（3）调用线程对象的start()方法来启动该线程。

```
public class FirstThreadTest extends Thread {
    int i = 0;

    //重写run方法，run方法的方法体就是现场执行体
    public void run() {
        for (; i < 100; i++) {
            System.out.println(getName() + "  " + i);

        }
    }

    public static void main(String[] args) {
        for (int i = 0; i < 100; i++) {
            System.out.println(Thread.currentThread().getName()
+ "  : " + i);
            if (i == 20) {
                new FirstThreadTest().start();
                new FirstThreadTest().start();
            }
        }
    }

}
```

上述代码中Thread.currentThread()方法返回当前正在执行的线程对象。GetName()方法返回调用该方法的线程的名字。

二、通过Runnable接口创建线程类

（1）定义runnable接口的实现类，并重写该接口的run()方法，该run()方法的方法体同样是该线程的线程执行体。

（2）创建 Runnable实现类的实例，并依此实例作为Thread的target来创建Thread对象，该Thread对象才是真正的线程对象。

（3）调用线程对象的start()方法来启动该线程。

```
public class RunnableThreadTest implements Runnable {
    private int i;

    public void run() {
        for (i = 0; i < 100; i++) {
            System.out.println(Thread.currentThread().getName()
+ " " + i);
        }
    }

    public static void main(String[] args) {
        for (int i = 0; i < 100; i++) {
            System.out.println(Thread.currentThread().getName()
+ " " + i);
            if (i == 20) {
                RunnableThreadTest rtt = new RunnableThreadTest(
);
                new Thread(rtt, "新线程1").start();
                new Thread(rtt, "新线程2").start();
            }
        }

    }

}
```

三、通过Callable和Future创建线程

（1）创建Callable接口的实现类，并实现call()方法，该call()方法将作为线程执行体，并且有返回值。

（2）创建Callable实现类的实例，使用FutureTask类来包装Callable对象，该FutureTask对象封装了该Callable对象的call()方法的返回值。

（3）使用FutureTask对象作为Thread对象的target创建并启动新线程。

（4）调用FutureTask对象的get()方法来获得子线程执行结束后的返回值，调用get()方法会阻塞线程。

```
public class CallableThreadTest implements Callable<Integer> {

    public static void main(String[] args) {
        CallableThreadTest ctt = new CallableThreadTest();
        FutureTask<Integer> ft = new FutureTask<>(ctt);
        for (int i = 0; i < 100; i++) {
            System.out.println(Thread.currentThread().getName()
+ " 的循环变量i的值" + i);
            if (i == 20) {
                new Thread(ft, "有返回值的线程").start();
            }
        }
        try {
            System.out.println("子线程的返回值：" + ft.get());
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
    }

    @Override
    public Integer call() throws Exception {
        int i = 0;
        for (; i < 100; i++) {
            System.out.println(Thread.currentThread().getName()
+ " " + i);
        }
        return i;
    }
}
```

四、创建线程的三种方式的对比

采用实现**Runnable**、**Callable**接口的方式创见多线程时，优势是：

线程类只是实现了Runnable接口或Callable接口，还可以继承其他类。

在这种方式下，多个线程可以共享同一个target对象，所以非常适合多个相同线程来处理同一份资源的情况，从而可以将CPU、代码和数据分开，形成清晰的模型，较好地体现了面向对象的思想。

劣势是：

编程稍微复杂，如果要访问当前线程，则必须使用Thread.currentThread()方法。

使用继承**Thread**类的方式创建多线程时优势是：

编写简单，如果需要访问当前线程，则无需使用Thread.currentThread()方法，直接使用this即可获得当前线程。

劣势是：

线程类已经继承了Thread类，所以不能再继承其他父类。

一、概述

在我们的开发中经常会使用到多线程。例如在Android中，由于主线程的诸多限制，像网络请求等一些耗时的操作我们必须在子线程中运行。我们往往会通过new Thread来开启一个子线程，待子线程操作完成以后通过Handler切换到主线程中运行。这么以来我们无法管理我们所创建的子线程，并且无限制的创建子线程，它们相互之间竞争，很有可能由于占用过多资源而导致死机或者OOM。所以在Java中为我们提供了线程池来管理我们所创建的线程。

线程池的优势

①降低系统资源消耗，通过重用已存在的线程，降低线程创建和销毁造成的消耗；

②提高系统响应速度，当有任务到达时，无需等待新线程的创建便能立即执行；

③方便线程并发数的管控，线程若是无限制的创建，不仅会额外消耗大量系统资源，更是占用过多资源而阻塞系统或oom等状况，从而降低系统的稳定性。线程池能有效管控线程，统一分配、调优，提供资源使用率；

④更强大的功能，线程池提供了定时、定期以及可控线程数等功能的线程池，使用方便简单。

二、ThreadPoolExecutor

我们可以通过ThreadPoolExecutor来创建一个线程池。

```
ExecutorService service = new ThreadPoolExecutor(....);
```

下面我们就来看一下ThreadPoolExecutor中的一个构造方法。

```
public ThreadPoolExecutor(int corePoolSize,
    int maximumPoolSize,
    long keepAliveTime,
    TimeUnit unit,
    BlockingQueue<Runnable> workQueue,
    ThreadFactory threadFactory,
    RejectedExecutionHandler handler)
```

ThreadPoolExecutor参数含义

1. corePoolSize

线程池中的核心线程数，默认情况下，核心线程一直存活在线程池中，即便他们在线程池中处于闲置状态。除非我们将ThreadPoolExecutor的allowCoreThreadTimeOut属性设为true的时候，这时候处于闲置的核心线程在等待新任务到来时会有超时策略，这个超时时间由keepAliveTime来指定。一旦超过所设置的超时时间，闲置的核心线程就会被终止。

2. maximumPoolSize

线程池中所容纳的最大线程数，如果活动的线程达到这个数值以后，后续的新任务将会被阻塞。包含核心线程数+非核心线程数。

3. keepAliveTime

非核心线程闲置时的超时时长，对于非核心线程，闲置时间超过这个时间，非核心线程就会被回收。只有对ThreadPoolExecutor的allowCoreThreadTimeOut属性设为true的时候，这个超时时间才会对核心线程产生效果。

4. unit

用于指定keepAliveTime参数的时间单位。他是一个枚举，可以使用的单位有天（TimeUnit.DAYS），小时（TimeUnit.HOURS），分钟（TimeUnit.MINUTES），毫秒(TimeUnit.MILLISECONDS)，微秒(TimeUnit.MICROSECONDS, 千分之一毫秒)和毫微秒(TimeUnit.NANOSECONDS, 千分之一微秒);

5. workQueue

线程池中保存等待执行的任务的阻塞队列。通过线程池中的execute方法提交的Runable对象都会存储在该队列中。我们可以选择下面几个阻塞队列。

阻塞队列	说明
ArrayBlockingQueue	基于数组实现的有界的阻塞队列,该队列按照FIFO（先进先出）原则对队列中的元素进行排序。
LinkedBlockingQueue	基于链表实现的阻塞队列，该队列按照FIFO（先进先出）原则对队列中的元素进行排序。
SynchronousQueue	内部没有任何容量的阻塞队列。在它内部没有任何的缓存空间。对于SynchronousQueue中的数据元素只有当我们试着取走的时候才可能存在。
PriorityBlockingQueue	具有优先级的无限阻塞队列。

我们还能够通过实现BlockingQueue接口来自定义我们所需要的阻塞队列。

6. threadFactory

线程工厂，为线程池提供新线程的创建。ThreadFactory是一个接口，里面只有一个newThread方法。 默认为DefaultThreadFactory类。

7. handler

是RejectedExecutionHandler对象，而RejectedExecutionHandler是一个接口，里面只有一个rejectedExecution方法。当任务队列已满并且线程池中的活动线程已经达到所限定的最大值或者是无法成功执行任务，这时候**ThreadPoolExecutor**会调用**RejectedExecutionHandler**中的**rejectedExecution**方法。在ThreadPoolExecutor中有四个内部类实现了RejectedExecutionHandler接口。在线程池中它默认是**AbortPolicy**，在无法处理新任务时抛出**RejectedExecutionException**异常。

下面是在ThreadPoolExecutor中提供的四个可选值。

可选值	说明
CallerRunsPolicy	只用调用者所在线程来运行任务。
AbortPolicy	直接抛出RejectedExecutionException异常。
DiscardPolicy	丢弃掉该任务，不进行处理。
DiscardOldestPolicy	丢弃队列里最近的一个任务，并执行当前任务。

我们也可以通过实现RejectedExecutionHandler接口来自定义我们自己的handler。如记录日志或持久化不能处理的任务。

ThreadPoolExecutor的使用

```
ExecutorService service = new ThreadPoolExecutor(5, 10, 10, Time
Unit.SECONDS, new LinkedBlockingQueue<>());
```

对于ThreadPoolExecutor有多个构造方法，对于上面的构造方法中的其他参数都采用默认值。可以通过execute和submit两种方式来向线程池提交一个任务。
execute 当我们使用execute来提交任务时，由于execute方法没有返回值，所以说我们也就无法判定任务是否被线程池执行成功。

```
service.execute(new Runnable() {
    public void run() {
        System.out.println("execute方式");
    }
});
```

submit

当我们使用submit来提交任务时,它会返回一个future,我们就可以通过这个future来判断任务是否执行成功，还可以通过future的get方法来获取返回值。如果子线程任务没有完成，get方法会阻塞住直到任务完成，而使用get(long timeout, TimeUnit unit)方法则会阻塞一段时间后立即返回，这时候有可能任务并没有执行完。

```
Future<Integer> future = service.submit(new Callable<Integer>()
{

    @Override
    public Integer call() throws Exception {
        System.out.println("submit方式");
        return 2;
    }
});
try {
    Integer number = future.get();
} catch (ExecutionException e) {
                // TODO Auto-generated catch block
    e.printStackTrace();
}
```

线程池关闭

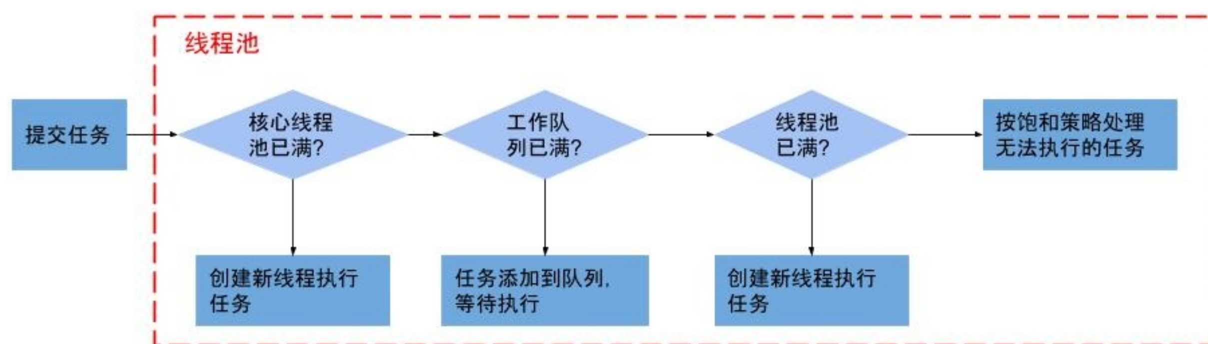
调用线程池的 `shutdown()` 或 `shutdownNow()` 方法来关闭线程池

shutdown原理：将线程池状态设置成SHUTDOWN状态，然后中断所有没有正在执行任务的线程。

shutdownNow原理：将线程池的状态设置成STOP状态，然后中断所有任务(包括正在执行的)的线程，并返回等待执行任务的列表。

中断采用**interrupt**方法，所以无法响应中断的任务可能永远无法终止。 但调用上述的两个关闭之一，isShutdown()方法返回值为true，当所有任务都已关闭，表示线程池关闭完成，则isTerminated()方法返回值为true。当需要立刻中断所有的线程，不一定需要执行完任务，可直接调用shutdownNow()方法。

三、线程池执行流程



①如果在线程池中的线程数量没有达到核心的线程数量，这时候就回启动一个核心线程来执行任务。

②如果线程池中的线程数量已经超过核心线程数，这时候任务就会被插入到任务队列中排队等待执行。

③由于任务队列已满，无法将任务插入到任务队列中。这个时候如果线程池中的线程数量没有达到线程池所设定的最大值，那么这时候就会立即启动一个非核心线程来执行任务。

④如果线程池中的数量达到了所规定的最大值，那么就会拒绝执行此任务，这时候就会调用RejectedExecutionHandler中的rejectedExecution方法来通知调用者。

四、四种线程池类

Java中四种具有不同功能常见的线程池。他们都是直接或者间接配置ThreadPoolExecutor来实现他们各自的功能。这四种线程池分别是newFixedThreadPool,newCachedThreadPool,newScheduledThreadPool和newSingleThreadExecutor。这四个线程池可以通过Executors类获取。

1. newFixedThreadPool

通过Executors中的newFixedThreadPool方法来创建，该线程池是一种线程数量固定的线程池。

```
ExecutorService service = Executors.newFixedThreadPool(4);
```

在这个线程池中 所容纳最大的线程数就是我们设置的核心线程数。 如果线程池的线程处于空闲状态的话，它们并不会被回收，除非是这个线程池被关闭。如果所有的线程都处于活动状态的话，新任务就会处于等待状态，直到有线程空闲出来。

由于newFixedThreadPool只有核心线程，并且这些线程都不会被回收，也就是 它能够更快速的响应外界请求 。从下面的newFixedThreadPool方法的实现可以看出，newFixedThreadPool只有核心线程，并且不存在超时机制，采用LinkedBlockingQueue，所以对于任务队列的大小也是没有限制的。

```
public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads,
        0L, TimeUnit.MILLISECONDS,
        new LinkedBlockingQueue<Runnable>());
}
```

2. newCachedThreadPool

通过Executors中的newCachedThreadPool方法来创建。

```
public static ExecutorService newCachedThreadPool() {
    return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
        60L, TimeUnit.SECONDS,
        new SynchronousQueue<Runnable>());
}
```

通过s上面的newCachedThreadPool方法在这里我们可以看出它的 核心线程数为 **0**， 线程池的最大线程数Integer.MAX_VALUE。而Integer.MAX_VALUE是一个很大的数，也差不多可以说 这个线程池中的最大线程数可以任意大。

当线程池中的线程都处于活动状态的时候，线程池就会创建一个新的线程来处理任务。该线程池中的线程超时时长为**60**秒，所以当线程处于闲置状态超过**60**秒的时候便会被回收。 这也就意味着若是整个线程池的线程都处于闲置状态超过60秒以后，在newCachedThreadPool线程池中是不存在任何线程的，所以这时候它几乎不占用任何的系统资源。

对于newCachedThreadPool他的任务队列采用的是SynchronousQueue，上面说到在SynchronousQueue内部没有任何容量的阻塞队列。SynchronousQueue内部相当于一个空集合，我们无法将一个任务插入到SynchronousQueue中。所以说在线程池中如果现有线程无法接收任务,将会创建新的线程来执行任务。

3. newScheduledThreadPool

通过Executors中的newScheduledThreadPool方法来创建。

```
public static ScheduledExecutorService newScheduledThreadPool(int
 corePoolSize) {
    return new ScheduledThreadPoolExecutor(corePoolSize);
}
public ScheduledThreadPoolExecutor(int corePoolSize) {
    super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS,
          new DelayedWorkQueue());
}
```

它的核心线程数是固定的，对于非核心线程几乎可以说是没有限制的，并且当非核心线程处于限制状态的时候就会立即被回收。

创建一个可定时执行或周期执行任务的线程池：

```
ScheduledExecutorService service = Executors.newScheduledThreadPool(4);
service.schedule(new Runnable() {
    public void run() {
        System.out.println(Thread.currentThread().getName()+"延迟三秒执行");
    }
}, 3, TimeUnit.SECONDS);
service.scheduleAtFixedRate(new Runnable() {
    public void run() {
        System.out.println(Thread.currentThread().getName()+"延迟三秒后每隔2秒执行");
    }
}, 3, 2, TimeUnit.SECONDS);
```

输出结果：

> pool-1-thread-2延迟三秒后每隔2秒执行
> pool-1-thread-1延迟三秒执行
> pool-1-thread-1延迟三秒后每隔2秒执行
> pool-1-thread-2延迟三秒后每隔2秒执行
> pool-1-thread-2延迟三秒后每隔2秒执行

`schedule(Runnable command, long delay, TimeUnit unit)`：延迟一定时间后执行Runnable任务；

`schedule(Callable callable, long delay, TimeUnit unit)`：延迟一定时间后执行Callable任务；

`scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit)`：延迟一定时间后，以间隔period时间的频率周期性地执行任务；

`scheduleWithFixedDelay(Runnable command, long initialDelay, long delay,TimeUnit unit)`:与scheduleAtFixedRate()方法很类似，但是不同的是scheduleWithFixedDelay()方法的周期时间间隔是以上一个任务执行结束到下一个任务开始执行的间隔，而scheduleAtFixedRate()方法的周期时间间隔是以上一个任务开始执行到下一个任务开始执行的间隔，也就是这一些任务系列的触发时间都是可预知的。

ScheduledExecutorService功能强大，对于定时执行的任务，建议多采用该方法。

4. newSingleThreadExecutor

通过Executors中的newSingleThreadExecutor方法来创建，在这个线程池中只有一个核心线程，对于任务队列没有大小限制，也就意味着这一个任务处于活动状态时，其他任务都会在任务队列中排队等候依次执行。

newSingleThreadExecutor将所有的外界任务统一到一个线程中支持，所以在这个任务执行之间我们不需要处理线程同步的问题。

```
public static ExecutorService newSingleThreadExecutor() {
    return new FinalizableDelegatedExecutorService
    (new ThreadPoolExecutor(1, 1,
        0L, TimeUnit.MILLISECONDS,
        new LinkedBlockingQueue<Runnable>()));
}
```

五、线程池的使用技巧

需要针对具体情况而具体处理，不同的任务类别应采用不同规模的线程池，任务类别可划分为CPU密集型任务、IO密集型任务和混合型任务。(N代表CPU个数)

任务类别	说明
CPU密集型任务	线程池中线程个数应尽量少，如配置N+1个线程的线程池。
IO密集型任务	由于IO操作速度远低于CPU速度，那么在运行这类任务时，CPU绝大多数时间处于空闲状态，那么线程池可以配置尽量多些的线程，以提高CPU利用率，如2*N。
混合型任务	可以拆分为CPU密集型任务和IO密集型任务，当这两类任务执行时间相差无几时，通过拆分再执行的吞吐率高于串行执行的吞吐率，但若这两类任务执行时间有数据级的差距，那么没有拆分的意义。

一、死锁产生的条件

一般来说，要出现死锁问题需要满足以下条件：

1. 互斥条件：一个资源每次只能被一个线程使用。
2. 请求与保持条件：一个线程因请求资源而阻塞时，对已获得的资源保持不放。
3. 不剥夺条件：线程已获得的资源，在未使用完之前，不能强行剥夺。
4. 循环等待条件：若干线程之间形成一种头尾相接的循环等待资源关系。

在JAVA编程中，有3种典型的死锁类型：
静态的锁顺序死锁，动态的锁顺序死锁，协作对象之间发生的死锁。

二、静态的锁顺序死锁

a和b两个方法都需要获得A锁和B锁。一个线程执行a方法且已经获得了A锁，在等待B锁；另一个线程执行了b方法且已经获得了B锁，在等待A锁。这种状态，就是发生了静态的锁顺序死锁。

```
//可能发生静态锁顺序死锁的代码
class StaticLockOrderDeadLock {
    private final Object lockA = new Object();
    private final Object lockB = new Object();

    public void a() {
        synchronized (lockA) {
            synchronized (lockB) {
                System.out.println("function a");
            }
        }
    }

    public void b() {
        synchronized (lockB) {
            synchronized (lockA) {
                System.out.println("function b");
            }
        }
    }
}
```

解决静态的锁顺序死锁的方法就是：所有需要多个锁的线程，都要以相同的顺序来获得锁。

```
//正确的代码
class StaticLockOrderDeadLock {
    private final Object lockA = new Object();
    private final Object lockB = new Object();

    public void a() {
        synchronized (lockA) {
            synchronized (lockB) {
                System.out.println("function a");
            }
        }
    }

    public void b() {
        synchronized (lockA) {
            synchronized (lockB) {
                System.out.println("function b");
            }
        }
    }
}
```

三、动态的锁顺序死锁：

动态的锁顺序死锁是指两个线程调用同一个方法时，传入的参数颠倒造成的死锁。如下代码，一个线程调用了transferMoney方法并传入参数accountA,accountB；另一个线程调用了transferMoney方法并传入参数accountB,accountA。此时就可能发生在静态的锁顺序死锁中存在的问题，即：第一个线程获得了accountA锁并等待accountB锁，第二个线程获得了accountB锁并等待accountA锁。

```
//可能发生动态锁顺序死锁的代码
class DynamicLockOrderDeadLock {
    public void transefMoney(Account fromAccount, Account toAcco
unt, Double amount) {
        synchronized (fromAccount) {
            synchronized (toAccount) {
                //...
                fromAccount.minus(amount);
                toAccount.add(amount);
                //...
            }
        }
    }
}
```

动态的锁顺序死锁解决方案如下：使用**System.identifyHashCode**来定义锁的顺序。确保所有的线程都以相同的顺序获得锁。

```
//正确的代码
class DynamicLockOrderDeadLock {
    private final Object myLock = new Object();

    public void transefMoney(final Account fromAccount, final Ac
count toAccount, final Double amount) {
        class Helper {
            public void transfer() {
                //...
                fromAccount.minus(amount);
                toAccount.add(amount);
                //...
            }
        }
        int fromHash = System.identityHashCode(fromAccount);
        int toHash = System.identityHashCode(toAccount);

        if (fromHash < toHash) {
            synchronized (fromAccount) {
                synchronized (toAccount) {
                    new Helper().transfer();
```

```
                }
            }
        } else if (fromHash > toHash) {
            synchronized (toAccount) {
                synchronized (fromAccount) {
                    new Helper().transfer();
                }
            }
        } else {
            synchronized (myLock) {
                synchronized (fromAccount) {
                    synchronized (toAccount) {
                        new Helper().transfer();
                    }
                }
            }
        }

    }
}
```

四、协作对象之间发生的死锁：

有时，死锁并不会那么明显，比如两个相互协作的类之间的死锁，比如下面的代码：一个线程调用了Taxi对象的setLocation方法，另一个线程调用了Dispatcher对象的getImage方法。此时可能会发生，第一个线程持有Taxi对象锁并等待Dispatcher对象锁，另一个线程持有Dispatcher对象锁并等待Taxi对象锁。

```
//可能发生死锁
class Taxi {
    private Point location, destination;
    private final Dispatcher dispatcher;

    public Taxi(Dispatcher dispatcher) {
        this.dispatcher = dispatcher;
    }

    public synchronized Point getLocation() {
```

```
        return location;
    }

    public synchronized void setLocation(Point location) {
        this.location = location;
        if (location.equals(destination))
            dispatcher.notifyAvailable(this);//外部调用方法，可能等
待Dispatcher对象锁
    }
}

class Dispatcher {
    private final Set<Taxi> taxis;
    private final Set<Taxi> availableTaxis;

    public Dispatcher() {
        taxis = new HashSet<Taxi>();
        availableTaxis = new HashSet<Taxi>();
    }

    public synchronized void notifyAvailable(Taxi taxi) {
        availableTaxis.add(taxi);
    }

    public synchronized Image getImage() {
        Image image = new Image();
        for (Taxi t : taxis)
            image.drawMarker(t.getLocation());//外部调用方法，可能
等待Taxi对象锁
        return image;
    }
}
```

上面的代码中， 我们在持有锁的情况下调用了外部的方法，这是非常危险的（可能发生死锁）。为了避免这种危险的情况发生， 我们使用开放调用。如果调用某个外部方法时不需要持有锁，我们称之为开放调用。

解决协作对象之间发生的死锁：需要使用开放调用，即避免在持有锁的情况下调用外部的方法。

```
//正确的代码
class Taxi {
    private Point location, destination;
    private final Dispatcher dispatcher;

    public Taxi(Dispatcher dispatcher) {
        this.dispatcher = dispatcher;
    }

    public synchronized Point getLocation() {
        return location;
    }

    public void setLocation(Point location) {
        boolean flag = false;
        synchronized (this) {
            this.location = location;
            flag = location.equals(destination);
        }
        if (flag)
            dispatcher.notifyAvailable(this);//使用开放调用
    }
}

class Dispatcher {
    private final Set<Taxi> taxis;
    private final Set<Taxi> availableTaxis;

    public Dispatcher() {
        taxis = new HashSet<Taxi>();
        availableTaxis = new HashSet<Taxi>();
    }

    public synchronized void notifyAvailable(Taxi taxi) {
        availableTaxis.add(taxi);
    }

    public Image getImage() {
        Set<Taxi> copy;
        synchronized (this) {
```

```
            copy = new HashSet<Taxi>(taxis);
        }
        Image image = new Image();
        for (Taxi t : copy)
            image.drawMarker(t.getLocation());//使用开放调用
        return image;
    }
}
```

五、总结

综上，是常见的3种死锁的类型。即：静态的锁顺序死锁，动态的锁顺序死锁，协作对象之间的死锁。在写代码时，要确保线程在获取多个锁时采用一致的顺序。同时，要避免在持有锁的情况下调用外部方法。

一、线程同步问题的产生及解决方案

问题的产生：

Java允许多线程并发控制，当多个线程同时操作一个可共享的资源变量时（如数据的增删改查），将会导致数据不准确，相互之间产生冲突。

如下例：假设有一个卖票系统，一共有100张票，有4个窗口同时卖。

```
public class Ticket implements Runnable {
    // 当前拥有的票数
    private int num = 100;

    public void run() {
        while (true) {
            if (num > 0) {
                try {
                    Thread.sleep(10);
                } catch (InterruptedException e) {
                }
                // 输出卖票信息
                System.out.println(Thread.currentThread().getNam
e() + ".....sale...." + num--);
            }
        }
    }
}
```

```
public class Nothing {

    public static void main(String[] args) {
        Ticket t = new Ticket();//创建一个线程任务对象。
        //创建4个线程同时卖票
        Thread t1 = new Thread(t);
        Thread t2 = new Thread(t);
        Thread t3 = new Thread(t);
        Thread t4 = new Thread(t);
        //启动线程
        t1.start();
        t2.start();
        t3.start();
        t4.start();
    }
}
```

输出部分结果：

> Thread-1.....sale....2
> Thread-0.....sale....3
> Thread-2.....sale....1
> Thread-0.....sale....0
> Thread-1.....sale....0
> Thread-3.....sale....1

显然上述结果是不合理的，对于同一张票进行了多次售出。这就是多线程情况下，出现了数据“脏读”情况。即多个线程访问余票num时，当一个线程获得余票的数量，要在此基础上进行-1的操作之前，其他线程可能已经卖出多张票，导致获得的num不是最新的，然后-1后更新的数据就会有误。这就需要线程同步的实现了。

问题的解决：

因此加入同步锁以避免在该线程没有完成操作之前，被其他线程的调用，从而保证了该变量的唯一性和准确性。

一共有两种锁，来实现线程同步问题，分别是：`synchronized` 和 `ReentrantLock` 。下面我们就带着上述问题，看看这两种锁是如何解决的。

二、synchronized关键字

1.synchronized简介

1. synchronized实现同步的基础：Java中每个对象都可以作为锁。当线程试图访问同步代码时，必须先获得对象锁，退出或抛出异常时必须释放锁。
2. Synchronzied实现同步的表现形式分为：代码块同步 和 方法同步。

2.synchronized原理

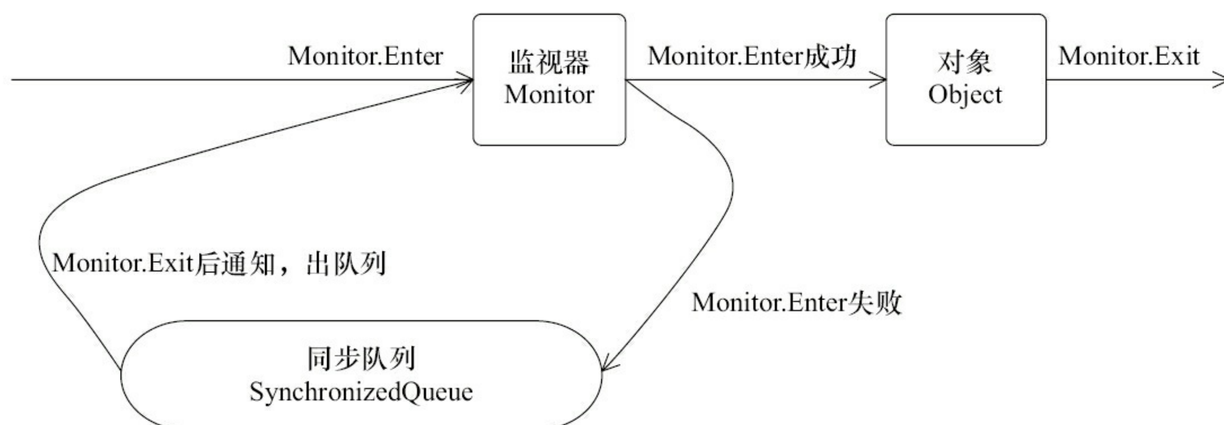
JVM基于进入和退出 `Monitor` 对象来实现 代码块同步 和 方法同步 ，两者实现细节不同。

代码块同步： 在编译后通过将 `monitorenter` 指令插入到同步代码块的开始处，将 `monitorexit` 指令插入到方法结束处和异常处，通过反编译字节码可以观察到。任何一个对象都有一个 `monitor` 与之关联，线程执行 `monitorenter` 指令时，会尝试获取对象对应的 `monitor` 的所有权，即尝试获得对象的锁。

方法同步： synchronized方法在 `method_info结构` 有 `ACC_synchronized` 标记，线程执行时会识别该标记，获取对应的锁，实现方法同步。

两者虽然实现细节不同，但本质上都是对一个对象的监视器（monitor）的获取。任意一个对象都拥有自己的监视器，当同步代码块或同步方法时，执行方法的线程必须先获得该对象的监视器才能进入同步块或同步方法，没有获取到监视器的线程将会被阻塞，并进入同步队列，状态变为 `BLOCKED` 。当成功获取监视器的线程释放了锁后，会唤醒阻塞在同步队列的线程，使其重新尝试对监视器的获取。

对象、监视器、同步队列和执行线程间的关系如下图：



3.synchronized的使用场景

①方法同步

```
public synchronized void method1
```

锁住的是该对象,类的其中一个实例，当该对象(仅仅是这一个对象)在不同线程中执行这个同步方法时，线程之间会形成互斥。达到同步效果，但如果不同线程同时对该类的不同对象执行这个同步方法时，则线程之间不会形成互斥，因为他们拥有的是不同的锁。

②代码块同步

```
synchronized(this){ //TODO }
```

描述同①

③方法同步

```
public synchronized static void method3
```

锁住的是该类，当所有该类的对象(多个对象)在不同线程中调用这个static同步方法时，线程之间会形成互斥，达到同步效果。

④代码块同步

```
synchronized(Test.class){ //TODO}
```

同③

⑤代码块同步

```
synchronized(o) {}
```

这里面的o可以是一个任何Object对象或数组，并不一定是它本身对象或者类，谁拥有o这个锁，谁就能够操作该块程序代码。

4.解决线程同步的实例

针对上述方法，具体的解决方式如下：

```
public class Ticket implements Runnable {
    // 当前拥有的票数
    private int num = 100;

    public void run() {
        while (true) {
            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {
            }
            synchronized (this) {
                // 输出卖票信息
                if (num > 0) {
                    System.out.println(Thread.currentThread().ge
tName() + ".....sale...." + num--);
                }

            }
        }
    }
}
```

输出部分结果：

> Thread-2.....sale....10
> Thread-1.....sale....9
> Thread-3.....sale....8
> Thread-0.....sale....7
> Thread-2.....sale....6
> Thread-1.....sale....5
> Thread-2.....sale....4
> Thread-1.....sale....3
> Thread-3.....sale....2
> Thread-0.....sale....1

可以看出实现了线程同步。同时改了一下逻辑，在进入到同步代码块时，先判断现在是否有没有票，然后再买票，防止出现没票还要售出的情况。通过同步代码块实现了线程同步，其他方法也一样可以实现该效果。

三、ReentrantLock锁

ReentrantLock，一个可重入的互斥锁，它具有与使用synchronized方法和语句所访问的隐式监视器锁相同的一些基本行为和语义，但功能更强大。（重入锁后面介绍）

1.Lock接口

Lock，锁对象。在Java中锁是用来控制多个线程访问共享资源的方式，一般来说，一个锁能够防止多个线程同时访问共享资源（但有的锁可以允许多个线程并发访问共享资源，比如读写锁，后面我们会分析）。在Lock接口出现之前，Java程序是靠 `synchronized` 关键字（后面分析）实现锁功能的，而JAVA SE5.0之后并发包中新增了 `Lock` 接口用来实现锁的功能，它提供了与 `synchronized` 关键字类似的同步功能，只是在使用时需要显式地获取和释放锁，缺点就是缺少像 `synchronized` 那样隐式获取释放锁的便捷性，但是却拥有了锁获取与释放的可操作性，可中断的获取锁以及超时获取锁等多种 `synchronized` 关键字所不具备的同步特性。

Lock接口的主要方法（还有两个方法比较复杂，暂不介绍）：

> **void lock():** 执行此方法时，如果锁处于空闲状态，当前线程将获取到锁。相反，如果锁已经被其他线程持有，将禁用当前线程，直到当前线程获取到锁。
> **boolean tryLock()**：如果锁可用，则获取锁，并立即返回true，否则返回false. 该方法和lock()的区别在于，tryLock()只是"试图"获取锁，如果锁不可用，不会导致当前线程被禁用，当前线程仍然继续往下执行代码。而lock()方法则是一定要获取到锁，如果锁不可用，就一直等待，在未获得锁之前,当前线程并不继续向下执行. 通常采用如下的代码形式调用tryLock()方法：
> **void unlock()**：执行此方法时，当前线程将释放持有的锁. 锁只能由持有者释放，如果线程并不持有锁，却执行该方法，可能导致异常的发生.
> **Condition newCondition()**：条件对象，获取等待通知组件。该组件和当前的锁绑定，当前线程只有获取了锁，才能调用该组件的await()方法，而调用后，当前线程将缩放锁。

2.ReentrantLock的使用

关于ReentrantLock的使用很简单，只需要显示调用，获得同步锁，释放同步锁即可。

```
ReentrantLock lock = new ReentrantLock(); //参数默认false，不公平锁
.....................
lock.lock(); //如果被其它资源锁定，会在此等待锁释放，达到暂停的效果
try {
    //操作
} finally {
    lock.unlock();  //释放锁
}
```

3.解决线程同步的实例

针对上述方法，具体的解决方式如下：

```
public class Ticket implements Runnable {
    // 当前拥有的票数
    private int num = 100;
    ReentrantLock lock = new ReentrantLock();

    public void run() {
        while (true) {
            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {
            }

            lock.lock();
            // 输出卖票信息
            if (num > 0) {
                System.out.println(Thread.currentThread().getNam
e() + ".....sale...." + num--);
            }
            lock.unlock();

        }
    }
}
```

四、重入锁

当一个线程得到一个对象后，再次请求该对象锁时是可以再次得到该对象的锁的。

具体概念就是：自己可以再次获取自己的内部锁。

Java里面内置锁(synchronized)和Lock(ReentrantLock)都是可重入的。

```
public class SynchronizedTest {
    public void method1() {
        synchronized (SynchronizedTest.class) {
            System.out.println("方法1获得ReentrantTest的锁运行了");
            method2();
        }
    }
    public void method2() {
        synchronized (SynchronizedTest.class) {
            System.out.println("方法1里面调用的方法2重入锁,也正常运行
了");
        }
    }
    public static void main(String[] args) {
        new SynchronizedTest().method1();
    }
}
```

上面便是synchronized的重入锁特性，即调用method1()方法时，已经获得了锁，此时内部调用method2()方法时，由于本身已经具有该锁，所以可以再次获取。

```
public class ReentrantLockTest {
    private Lock lock = new ReentrantLock();
    public void method1() {
        lock.lock();
        try {
            System.out.println("方法1获得ReentrantLock锁运行了");
            method2();
        } finally {
            lock.unlock();
        }
    }
    public void method2() {
        lock.lock();
        try {
            System.out.println("方法1里面调用的方法2重入ReentrantLoc
k锁,也正常运行了");
        } finally {
            lock.unlock();
        }
    }
    public static void main(String[] args) {
        new ReentrantLockTest().method1();
    }
}
```

上面便是ReentrantLock的重入锁特性，即调用method1()方法时，已经获得了锁，此时内部调用method2()方法时， 由于本身已经具有该锁，所以可以再次获取。

五、公平锁

CPU在调度线程的时候是在等待队列里随机挑选一个线程，由于这种随机性所以是无法保证线程先到先得的（synchronized控制的锁就是这种非公平锁）。但这样就会产生饥饿现象，即有些线程（优先级较低的线程）可能永远也无法获取CPU的执行权，优先级高的线程会不断的强制它的资源。那么如何解决饥饿问题呢，这就需要公平锁了。公平锁可以保证线程按照时间的先后顺序执行，避免饥饿现象的产生。但公平锁的效率比较低，因为要实现顺序执行，需要维护一个有序队列。

ReentrantLock便是一种公平锁，通过在构造方法中传入true就是公平锁，传入false，就是非公平锁。

```
public ReentrantLock(boolean fair) {
        sync = fair ? new FairSync() : new NonfairSync();
    }
```

以下是使用公平锁实现的效果：

```
public class LockFairTest implements Runnable{
    //创建公平锁
    private static ReentrantLock lock=new ReentrantLock(true);
    public void run() {
        while(true){
            lock.lock();
            try{
                System.out.println(Thread.currentThread().getNam
e()+"获得锁");
            }finally{
                lock.unlock();
            }
        }
    }
    public static void main(String[] args) {
        LockFairTest lft=new LockFairTest();
        Thread th1=new Thread(lft);
        Thread th2=new Thread(lft);
        th1.start();
        th2.start();
    }
}
```

输出结果：

> Thread-1获得锁
> Thread-0获得锁
> Thread-1获得锁
> Thread-0获得锁
> Thread-1获得锁
> Thread-0获得锁
> Thread-1获得锁
> Thread-0获得锁
> Thread-1获得锁
> Thread-0获得锁
> Thread-1获得锁
> Thread-0获得锁
> Thread-1获得锁
> Thread-0获得锁
> Thread-1获得锁
> Thread-0获得锁

这是截取的部分执行结果，分析结果可看出两个线程是交替执行的，几乎不会出现同一个线程连续执行多次。

六、synchronized和ReentrantLock的比较

1.区别：

1）Lock是一个接口，而synchronized是Java中的关键字，synchronized是内置的语言实现；

2）synchronized在发生异常时，会自动释放线程占有的锁，因此不会导致死锁现象发生；而Lock在发生异常时，如果没有主动通过unLock()去释放锁，则很可能造成死锁现象，因此使用Lock时需要在finally块中释放锁；

3）Lock可以让等待锁的线程响应中断，而synchronized却不行，使用synchronized时，等待的线程会一直等待下去，不能够响应中断；

4）通过Lock可以知道有没有成功获取锁，而synchronized却无法办到。

5）Lock可以提高多个线程进行读操作的效率。

总结：**ReentrantLock**相比**synchronized**，增加了一些高级的功能。但也有一定缺陷。

在ReentrantLock类中定义了很多方法，比如：

```
isFair()        //判断锁是否是公平锁

isLocked()      //判断锁是否被任何线程获取了

isHeldByCurrentThread()   //判断锁是否被当前线程获取了

hasQueuedThreads()   //判断是否有线程在等待该锁
```

2.两者在锁的相关概念上区别：

1)可中断锁

顾名思义，就是可以响应中断的锁。

在Java中，**synchronized**就不是可中断锁，而**Lock**是可中断锁。如果某一线程A正在执行锁中的代码，另一线程B正在等待获取该锁，可能由于等待时间过长，线程B不想等待了，想先处理其他事情，我们可以让它中断自己或者在别的线程中中断它，这种就是可中断锁。

`lockInterruptibly()` 的用法体现了Lock的可中断性。

2)公平锁

公平锁即尽量以请求锁的顺序来获取锁。比如同是有多个线程在等待一个锁，当这个锁被释放时，等待时间最久的线程（最先请求的线程）会获得该锁（并不是绝对的，大体上是这种顺序），这种就是公平锁。

非公平锁即无法保证锁的获取是按照请求锁的顺序进行的。这样就可能导致某个或者一些线程永远获取不到锁。

在Java中，synchronized就是非公平锁，它无法保证等待的线程获取锁的顺序。ReentrantLock可以设置成公平锁。

3)读写锁

读写锁将对一个资源（比如文件）的访问分成了2个锁，一个读锁和一个写锁。

正因为有了读写锁，才使得多个线程之间的读操作可以并发进行，不需要同步，而写操作需要同步进行，提高了效率。

ReadWriteLock就是读写锁，它是一个接口，ReentrantReadWriteLock实现了这个接口。

可以通过readLock()获取读锁，通过writeLock()获取写锁。

4)绑定多个条件

一个ReentrantLock对象可以同时绑定多个Condition对象，而在synchronized中，锁对象的wait()和notify()或notifyAll()方法可以实现一个隐含的条件，如果要和多余一个条件关联的时候，就不得不额外地添加一个锁，而ReentrantLock则无须这么做，只需要多次调用new Condition()方法即可。

3.性能比较

在性能上来说，如果竞争资源不激烈，两者的性能是差不多的，而 当竞争资源非常激烈时（即有大量线程同时竞争），此时**ReentrantLock**的性能要远远优于**synchronized** 。所以说，在具体使用时要根据适当情况选择。

在JDK1.5中，synchronized是性能低效的。因为这是一个重量级操作，它对性能最大的影响是阻塞的是实现，挂起线程和恢复线程的操作都需要转入内核态中完成，这些操作给系统的并发性带来了很大的压力。相比之下使用Java提供的ReentrankLock对象，性能更高一些。到了JDK1.6，发生了变化，对synchronize加入了很多优化措施，有自适应自旋，锁消除，锁粗化，轻量级锁，偏向锁等等。导致在JDK1.6上synchronize的性能并不比Lock差。官方也表示，他们也更支持synchronize，在未来的版本中还有优化余地，所以还是提倡在synchronized能实现需求的情况下，优先考虑使用synchronized来进行同步。

一、线程间通信的两种方式

1.wait()/notify()

Object类中相关的方法有notify方法和wait方法。因为wait和notify方法定义在Object类中，因此会被所有的类所继承。这些方法都是**final**的，即它们都是不能被重写的，不能通过子类覆写去改变它们的行为。

①wait()方法：让当前线程进入等待，并释放锁。

②wait(long)方法：让当前线程进入等待，并释放锁，不过等待时间为long，超过这个时间没有对当前线程进行唤醒，将自动唤醒。

③notify()方法：让当前线程通知那些处于等待状态的线程，当前线程执行完毕后释放锁，并从其他线程中唤醒其中一个继续执行。

④notifyAll()方法：让当前线程通知那些处于等待状态的线程，当前线程执行完毕后释放锁，将唤醒所有等待状态的线程。

wait()方法使用注意事项

①当前的线程必须拥有当前对象的monitor，也即lock，就是锁，才能调用wait()方法，否则将抛出异常java.lang.IllegalMonitorStateException。

②线程调用wait()方法，释放它对锁的拥有权，然后等待另外的线程来通知它（通知的方式是notify()或者notifyAll()方法），这样它才能重新获得锁的拥有权和恢复执行。

③要确保调用wait()方法的时候拥有锁，即，wait()方法的调用必须放在synchronized方法或synchronized块中。

wait()与**sleep()**比较

当线程调用了wait()方法时，它会释放掉对象的锁。

Thread.sleep()，它会导致线程睡眠指定的毫秒数，但线程在睡眠的过程中是不会释放掉对象的锁的。

notify()方法使用注意事项

①如果多个线程在等待，它们中的一个将会选择被唤醒。这种选择是随意的，和具体实现有关。（线程等待一个对象的锁是由于调用了wait()方法）。

②被唤醒的线程是不能被执行的，需要等到当前线程放弃这个对象的锁，当前线程会在方法执行完毕后释放锁。

wait()/notify()协作的两个注意事项

①通知过早

如果通知过早，则会打乱程序的运行逻辑。

```
public class MyRun {
    private String lock = new String("");
    public Runnable runnableA = new Runnable() {

        @Override
        public void run() {
            try {
                synchronized (lock) {
                    System.out.println("begin wait");
                    lock.wait();
                    System.out.println("end wait");
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

        }
    };
    public Runnable runnableB = new Runnable() {
        @Override
        public void run() {
            synchronized (lock) {
                System.out.println("begin notify");
                lock.notify();
                System.out.println("end notify");
            }
        }
    };
}
```

两个方法，分别执行wait()/notify()方法。

```java
public static void main(String[] args) throws InterruptedExcepti
on {
        MyRun run = new MyRun();
        Thread bThread = new Thread(run.runnableB);
        bThread.start();
        Thread.sleep(100);
        Thread aThread = new Thread(run.runnableA);
        aThread.start();
    }
```

如果notify()方法先执行，将导致wait()方法释放锁进入等待状态后，永远无法被唤醒，影响程序逻辑。应避免这种情况。

②等待wait的条件发生变化

在使用wait/notify模式时，还需要注意另外一种情况，也就是wait等待条件发生了变化，也容易造成程序逻辑的混乱。

Add类，执行加法操作，然后通知**Subtract**类

```java
public class Add {
    private String lock;

    public Add(String lock) {
        super();
        this.lock = lock;
    }
    public void add(){
        synchronized (lock) {
            ValueObject.list.add("anyThing");
            lock.notifyAll();
        }
    }
}
```

Subtract类，执行减法操作，执行完后进入等待状态，等待**Add**类唤醒**notify**

```
public class Subtract {
    private String lock;

    public Subtract(String lock) {
        super();
        this.lock = lock;
    }
    public void subtract(){
        try {
            synchronized (lock) {
                if(ValueObject.list.size()==0){
                    System.out.println("wait begin ThreadName="+
Thread.currentThread().getName());
                    lock.wait();
                    System.out.println("wait end ThreadName="+Th
read.currentThread().getName());
                }
                ValueObject.list.remove(0);
                System.out.println("list size ="+ValueObject.lis
t.size());
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

线程**ThreadAdd**

```
public class ThreadAdd extends Thread{
    private Add pAdd;

    public ThreadAdd(Add pAdd) {
        super();
        this.pAdd = pAdd;
    }
    @Override
    public void run() {
        pAdd.add();
    }

}
```

线程**ThreadSubtract**

```
public class ThreadSubtract extends Thread{
    private Subtract rSubtract;

    public ThreadSubtract(Subtract rSubtract) {
        super();
        this.rSubtract = rSubtract;
    }
    @Override
    public void run() {
        rSubtract.subtract();
    }

}
```

先开启两个**ThreadSubtract**线程，由于**list**中没有元素，进入等待状态。再开启一个**ThreadAdd**线程，向**list**中增加一个元素，然后唤醒两个**ThreadSubtract**线程。

```
public static void main(String[] args) throws InterruptedExcepti
on {
        String lock = new String("");
        Add add = new Add(lock);
        Subtract subtract = new Subtract(lock);
        ThreadSubtract subtractThread1 = new ThreadSubtract(subt
ract);
        subtractThread1.setName("subtractThread1");
        subtractThread1.start();
        ThreadSubtract subtractThread2 = new ThreadSubtract(subt
ract);
        subtractThread2.setName("subtractThread2");
        subtractThread2.start();
        Thread.sleep(1000);
        ThreadAdd addThread = new ThreadAdd(add);
        addThread.setName("addThread");
        addThread.start();
    }
```

输出结果

> wait begin ThreadName=subtractThread1
> wait begin ThreadName=subtractThread2
> wait end ThreadName=subtractThread2
> Exception in thread "subtractThread1" list size =0
> wait end ThreadName=subtractThread1
> java.lang.IndexOutOfBoundsException: Index: 0, Size: 0
> at java.util.ArrayList.rangeCheck(Unknown Source)
> at java.util.ArrayList.remove(Unknown Source)
> at com.lvr.communication.Subtract.subtract(Subtract.java:18)
> at com.lvr.communication.ThreadSubtract.run(ThreadSubtract.java:12)

当第二个**ThreadSubtract**线程执行减法操作时，抛出下标越界异常。

原因分析：一开始两个**ThreadSubtract**线程等待状态，当**ThreadAdd**线程添加一个元素并唤醒所有线程后，第一个**ThreadSubtract**线程接着原来的执行到的地点开始继续执行，删除一个元素并输出集合大小。同样，第二个**ThreadSubtract**线程也如此，可是此时集合中已经没有元素了，所以抛出异常。

解决办法：从等待状态被唤醒后，重新判断条件，看看是否扔需要进入等待状态，不需要进入再进行下一步操作。即把**if()**判断，改成**while()**。

```
public void subtract(){
        try {
            synchronized (lock) {
                while(ValueObject.list.size()==0){
                    System.out.println("wait begin ThreadName="+
Thread.currentThread().getName());
                    lock.wait();
                    System.out.println("wait end ThreadName="+Th
read.currentThread().getName());
                }
                ValueObject.list.remove(0);
                System.out.println("list size ="+ValueObject.lis
t.size());
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
```

这是线程间协作中经常出现的一种情况，需要避免。

2.Condition实现等待/通知

关键字synchronized与wait()和notify()/notifyAll()方法相结合可以实现等待/通知模式，类似ReentrantLock也可以实现同样的功能，但需要借助于Condition对象。

关于Condition实现等待/通知就不详细介绍了，可以完全类比wait()/notify()，基本使用和注意事项完全一致。
就只简单介绍下类比情况：

condition.await()————>lock.wait()

condition.await(long time, TimeUnit unit)————>lock.wait(long timeout)

condition.signal()————>lock.notify()

condition.signaAll()————>lock.notifyAll()

特殊之处：**synchronized**相当于整个**ReentrantLock**对象只有一个单一的**Condition**对象情况。而一个**ReentrantLock**却可以拥有多个**Condition**对象，来实现通知部分线程。

具体实现方式：
假设有两个Condition对象：ConditionA和ConditionB。那么由ConditionA.await()方法进入等待状态的线程，由ConditionA.signalAll()通知唤醒；由ConditionB.await()方法进入等待状态的线程，由ConditionB.signalAll()通知唤醒。篇幅有限，代码示例就不写了。

二、生产者/消费者模式实现

1.一生产与一消费

下面情形是一个生产者，一个消费者的模式。假设场景：一个String对象，其中生产者为其设置值，消费者拿走其中的值，不断的循环往复，实现生产者/消费者的情形。

wait()/notify()实现

生产者

```
public class Product {
    private String lock;

    public Product(String lock) {
        super();
        this.lock = lock;
    }
    public void setValue(){
        try {
            synchronized (lock) {
                if(!StringObject.value.equals("")){
                    //有值，不生产
                    lock.wait();
                }
                String  value = System.currentTimeMillis()+""+Sy
stem.nanoTime();
                System.out.println("set的值是："+value);
                StringObject.value = value;
                lock.notify();
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

消费者

```
public class Consumer {
    private String lock;

    public Consumer(String lock) {
        super();
        this.lock = lock;
    }
    public void getValue(){
        try {
            synchronized (lock) {
                if(StringObject.value.equals("")){
                    //没值，不进行消费
                    lock.wait();
                }
                System.out.println("get的值是："+StringObject.val
ue);
                StringObject.value = "";
                lock.notify();
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

生产者线程

```
public class ThreadProduct extends Thread{
    private Product product;

    public ThreadProduct(Product product) {
        super();
        this.product = product;
    }
    @Override
    public void run() {
        //死循环，不断的生产
        while(true){
            product.setValue();
        }
    }

}
```

消费者线程

```
public class ThreadConsumer extends Thread{
    private Consumer consumer;

    public ThreadConsumer(Consumer consumer) {
        super();
        this.consumer = consumer;
    }
    @Override
    public void run() {
        //死循环，不断的消费
        while(true){
            consumer.getValue();
        }
    }

}
```

开启生产者/消费者模式

```
public class Test {

    public static void main(String[] args) throws InterruptedExc
eption {
        String lock = new String("");
        Product product = new Product(lock);
        Consumer consumer = new Consumer(lock);
        ThreadProduct pThread = new ThreadProduct(product);
        ThreadConsumer cThread = new ThreadConsumer(consumer);
        pThread.start();
        cThread.start();
    }

}
```

输出结果：

> set的值是：148827033184127168687409691
> get的值是：148827033184127168687409691
> set的值是：148827033184127168687449887
> get的值是：148827033184127168687449887
> set的值是：148827033184127168687475117
> get的值是：148827033184127168687475117

Condition方式实现类似，篇幅有限不全部贴出来。

2.多生产与多消费

特殊情况： 按照上述一生产与一消费的情况，通过创建多个生产者和消费者线程，实现多生产与多消费的情况，将会出现“假死”。

具体原因： 多个生产者和消费者线程。当全部运行后，生产者线程生产数据后，可能唤醒的同类即生产者线程。此时可能会出现如下情况：所有生产者线程进入等待状态，然后消费者线程消费完数据后，再次唤醒的还是消费者线程，直至所有消费者线程都进入等待状态，此时将进入“假死”。

解决方法： 将notify()或signal()方法改为notifyAll()或signalAll()方法，这样就不怕因为唤醒同类而进入“假死”状态了。

Condition方式实现 生产者

```
public class Product {
    private ReentrantLock lock;
    private Condition condition;

    public Product(ReentrantLock lock, Condition condition) {
        super();
        this.lock = lock;
        this.condition = condition;
    }

    public void setValue() {
        try {
            lock.lock();
            while (!StringObject.value.equals("")) {
                // 有值，不生产
                condition.await();
            }
            String value = System.currentTimeMillis() + "" + Sys
tem.nanoTime();
            System.out.println("set的值是：" + value);
            StringObject.value = value;
            condition.signalAll();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }finally {
            lock.unlock();
        }


    }
}
```

消费者

```
public class Consumer {
    private ReentrantLock lock;
    private Condition condition;

    public Consumer(ReentrantLock lock,Condition condition) {
        super();
        this.lock = lock;
        this.condition = condition;
    }
    public void getValue(){
        try {
                lock.lock();
                while(StringObject.value.equals("")){
                    //没值，不进行消费
                    condition.await();
                }
                System.out.println("get的值是："+StringObject.val
ue);
                StringObject.value = "";
                condition.signalAll();

        } catch (InterruptedException e) {
            e.printStackTrace();
        }finally {
            lock.unlock();
        }
    }
}
```

生产者线程和消费者线程与一生产一消费的模式相同。

开启多生产/多消费模式

```
public static void main(String[] args) throws InterruptedExcepti
on {
        ReentrantLock lock = new ReentrantLock();
        Condition newCondition = lock.newCondition();
        Product product = new Product(lock,newCondition);
        Consumer consumer = new Consumer(lock,newCondition);
        for(int i=0;i<3;i++){
            ThreadProduct pThread = new ThreadProduct(product);
            ThreadConsumer cThread = new ThreadConsumer(consumer
);
            pThread.start();
            cThread.start();
        }

    }
```

输出结果:

> set的值是：148827212374628960540784817
> get的值是：148827212374628960540784817
> set的值是：148827212374628960540810047
> get的值是：148827212374628960540810047

可见交替地进行get/set实现多生产/多消费模式。

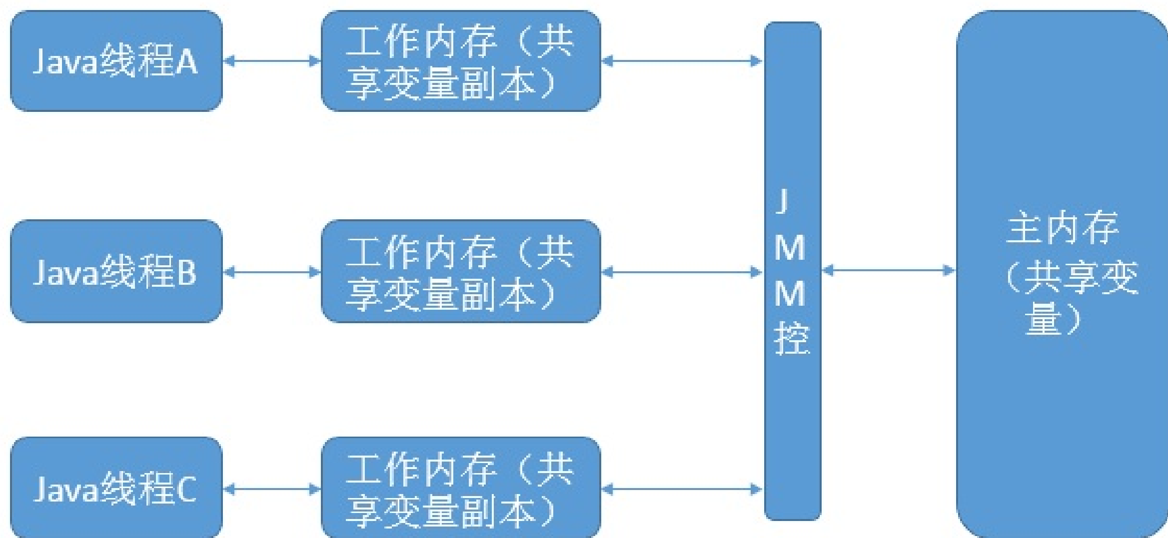
注意：相比一生产一消费的模式，改动了两处。①**signal()-->signalAll()**避免进入“假死”状态。②**if()**判断**-->while()**循环，重新判断条件，避免逻辑混乱。

以上就是Java线程间通信的相关知识，以生产者/消费者模式为例，讲解线程间通信的使用以及注意事项。

一、Java内存模型

想要理解 `volatile` 为什么能确保可见性，就要先理解Java中的内存模型是什么样的。

Java内存模型规定了所有的变量都存储在主内存中。每条线程中还有自己的工作内存，线程的工作内存中保存了被该线程所使用到的变量（这些变量是从主内存中拷贝而来）。线程对变量的所有操作（读取，赋值）都必须在工作内存中进行。不同线程之间也无法直接访问对方工作内存中的变量，线程间变量值的传递均需要通过主内存来完成。



基于此种内存模型，便产生了多线程编程中的数据“脏读”等问题。

举个简单的例子：在java中，执行下面这个语句：

```
i  = 10;
```

执行线程必须先在自己的工作线程中对变量i所在的缓存行进行赋值操作，然后再写入主存当中。而不是直接将数值**10**写入主存当中。

比如同时有2个线程执行这段代码，假如初始时i的值为10，那么我们希望两个线程执行完之后i的值变为12。但是事实会是这样吗？

可能存在下面一种情况：初始时，两个线程分别读取i的值存入各自所在的工作内存当中，然后线程1进行加1操作，然后把i的最新值11写入到内存。此时线程2的工作内存当中i的值还是10，进行加1操作之后，i的值为11，然后线程2把i的值写入内存。

最终结果i的值是11，而不是12。这就是著名的缓存一致性问题。通常称这种被多个线程访问的变量为共享变量。

那么如何确保共享变量在多线程访问时能够正确输出结果呢？

在解决这个问题之前，我们要先了解并发编程的三大概念：原子性，有序性，可见性。

二、原子性

1.定义

原子性：即一个操作或者多个操作，要么全部执行，并且执行的过程不会被任何因素打断，要么就都不执行。

2.实例

一个很经典的例子就是银行账户转账问题：

比如从账户A向账户B转1000元，那么必然包括2个操作：从账户A减去1000元，往账户B加上1000元。

试想一下，如果这2个操作不具备原子性，会造成什么样的后果。假如从账户A减去1000元之后，操作突然中止。这样就会导致账户A虽然减去了1000元，但是账户B没有收到这个转过来的1000元。

所以这2个操作必须要具备原子性才能保证不出现一些意外的问题。

同样地反映到并发编程中会出现什么结果呢？

举个最简单的例子，大家想一下假如为一个32位的变量赋值过程不具备原子性的话，会发生什么后果？

```
i = 9;
```

假若一个线程执行到这个语句时，我暂且假设为一个32位的变量赋值包括两个过程：为低16位赋值，为高16位赋值。

那么就可能发生一种情况：当将低16位数值写入之后，突然被中断，而此时又有一个线程去读取i的值，那么读取到的就是错误的数据。

3.Java中的原子性

在Java中，对基本数据类型的变量的读取和赋值操作是原子性操作，即这些操作是不可被中断的，要么执行，要么不执行。

上面一句话虽然看起来简单，但是理解起来并不是那么容易。看下面一个例子i：

请分析以下哪些操作是原子性操作：

```
x = 10;         //语句1
y = x;          //语句2
x++;            //语句3
x = x + 1;      //语句4
```

咋一看，可能会说上面的4个语句中的操作都是原子性操作。其实只有语句1是原子性操作，其他三个语句都不是原子性操作。

语句1是直接将数值10赋值给x，也就是说线程执行这个语句的会直接将数值10写入到工作内存中。

语句2实际上包含2个操作，它先要去读取x的值，再将x的值写入工作内存，虽然读取x的值以及将x的值写入工作内存，这2个操作都是原子性操作，但是合起来就不是原子性操作了。

同样的，x++和 x = x+1包括3个操作：读取x的值，进行加1操作，写入新的值。

所以上面4个语句只有语句1的操作具备原子性。

也就是说，只有简单的读取、赋值（而且必须是将数字赋值给某个变量，变量之间的相互赋值不是原子操作）才是原子操作。

从上面可以看出，Java内存模型只保证了基本读取和赋值是原子性操作，如果要实现更大范围操作的原子性，可以通过**synchronized**和**Lock**来实现。由于**synchronized**和**Lock**能够保证任一时刻只有一个线程执行该代码块，那么自然就

不存在原子性问题了，从而保证了原子性。

三、可见性

1.定义

可见性是指当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。

2.实例

举个简单的例子，看下面这段代码：

```
//线程1执行的代码
int i = 0;
i = 10;

//线程2执行的代码
j = i;
```

由上面的分析可知，当线程1执行 i =10这句时，会先把i的初始值加载到工作内存中，然后赋值为10，那么在线程1的工作内存当中i的值变为10了，却没有立即写入到主存当中。

此时线程2执行 j = i，它会先去主存读取i的值并加载到线程2的工作内存当中，注意此时内存当中i的值还是0，那么就会使得j的值为0，而不是10.

这就是可见性问题，线程1对变量i修改了之后，线程2没有立即看到线程1修改的值。

3.Java中的可见性

对于可见性，Java提供了volatile关键字来保证可见性。

当一个共享变量被**volatile**修饰时，它会保证修改的值会立即被更新到主存，当有其他线程需要读取时，它会去内存中读取新值。

而普通的共享变量不能保证可见性，因为普通共享变量被修改之后，什么时候被写入主存是不确定的，当其他线程去读取时，此时内存中可能还是原来的旧值，因此无法保证可见性。

另外，通过synchronized和Lock也能够保证可见性，synchronized和Lock能保证同一时刻只有一个线程获取锁然后执行同步代码，并且在释放锁之前会将对变量的修改刷新到主存当中。因此可以保证可见性。

四、有序性

1.定义

有序性：即程序执行的顺序按照代码的先后顺序执行。

2.实例

举个简单的例子，看下面这段代码：

```
int i = 0;

boolean flag = false;

i = 1;                //语句1
flag = true;          //语句2
```

上面代码定义了一个int型变量，定义了一个boolean类型变量，然后分别对两个变量进行赋值操作。从代码顺序上看，语句1是在语句2前面的，那么JVM在真正执行这段代码的时候会保证语句1一定会在语句2前面执行吗？不一定，为什么呢？这里可能会发生指令重排序（Instruction Reorder）。

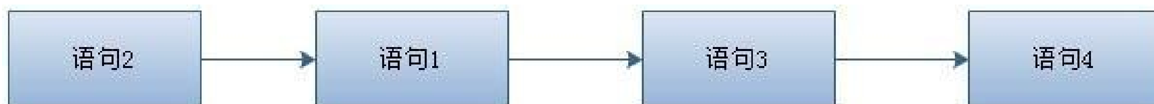
下面解释一下什么是指令重排序，一般来说，处理器为了提高程序运行效率，可能会对输入代码进行优化，它不保证程序中各个语句的执行先后顺序同代码中的顺序一致，但是它会保证程序最终执行结果和代码顺序执行的结果是一致的。

比如上面的代码中，语句1和语句2谁先执行对最终的程序结果并没有影响，那么就有可能在执行过程中，语句2先执行而语句1后执行。

但是要注意，虽然处理器会对指令进行重排序，但是它会保证程序最终结果会和代码顺序执行结果相同，那么它靠什么保证的呢？再看下面一个例子：

```
int a = 10;    //语句1
int r = 2;    //语句2
a = a + 3;    //语句3
r = a*a;     //语句4
```

这段代码有4个语句，那么可能的一个执行顺序是：



那么可不可能是这个执行顺序呢： 语句2 语句1 语句4 语句3

不可能，因为处理器在进行重排序时是会考虑指令之间的数据依赖性，如果一个指令**Instruction 2**必须用到**Instruction 1**的结果，那么处理器会保证**Instruction 1**会在**Instruction 2**之前执行。

虽然重排序不会影响单个线程内程序执行的结果，但是多线程呢？下面看一个例子：

```
//线程1:

context = loadContext();   //语句1
inited = true;             //语句2

 //线程2:
while(!inited ){
   sleep()
}
doSomethingwithconfig(context);
```

上面代码中，由于语句1和语句2没有数据依赖性，因此可能会被重排序。假如发生了重排序，在线程1执行过程中先执行语句2，而此是线程2会以为初始化工作已经完成，那么就会跳出while循环，去执行doSomethingwithconfig(context)方法，而此时context并没有被初始化，就会导致程序出错。

从上面可以看出，指令重排序不会影响单个线程的执行，但是会影响到线程并发执行的正确性。

也就是说，要想并发程序正确地执行，必须要保证原子性、可见性以及有序性。只要有一个没有被保证，就有可能会导致程序运行不正确。

3.Java中的有序性

在Java内存模型中，允许编译器和处理器对指令进行重排序，但是重排序过程不会影响到单线程程序的执行，却会影响到多线程并发执行的正确性。

在Java里面，可以通过volatile关键字来保证一定的“有序性”。另外可以通过synchronized和Lock来保证有序性，很显然，synchronized和Lock保证每个时刻是有一个线程执行同步代码，相当于是让线程顺序执行同步代码，自然就保证了有序性。

另外，Java内存模型具备一些先天的“有序性”，即不需要通过任何手段就能够得到保证的有序性，这个通常也称为 **happens-before** 原则。如果两个操作的执行次序无法从**happens-before**原则推导出来，那么它们就不能保证它们的有序性，虚拟机可以随意地对它们进行重排序。

下面就来具体介绍下**happens-before**原则（先行发生原则）：

①程序次序规则：一个线程内，按照代码顺序，书写在前面的操作先行发生于书写在后面的操作

②锁定规则：一个unLock操作先行发生于后面对同一个锁的lock操作

③volatile变量规则：对一个变量的写操作先行发生于后面对这个变量的读操作

④传递规则：如果操作A先行发生于操作B，而操作B又先行发生于操作C，则可以得出操作A先行发生于操作C

⑤线程启动规则：Thread对象的start()方法先行发生于此线程的每个一个动作

⑥线程中断规则：对线程interrupt()方法的调用先行发生于被中断线程的代码检测到中断事件的发生

⑦线程终结规则：线程中所有的操作都先行发生于线程的终止检测，我们可以通过Thread.join()方法结束、Thread.isAlive()的返回值手段检测到线程已经终止执行

⑧对象终结规则：一个对象的初始化完成先行发生于他的finalize()方法的开始

这8条规则中，前4条规则是比较重要的，后4条规则都是显而易见的。

下面我们来解释一下前4条规则：

对于程序次序规则来说，就是一段程序代码的执行在单个线程中看起来是有序的。注意，虽然这条规则中提到“书写在前面的操作先行发生于书写在后面的操作”，这个应该是程序看起来执行的顺序是按照代码顺序执行的，但是虚拟机可能会对程序代码进行指令重排序。虽然进行重排序，但是最终执行的结果是与程序顺序执行的结果一致的，它只会对不存在数据依赖性的指令进行重排序。因此，在单个线程中，程序执行看起来是有序执行的，这一点要注意理解。事实上，这个规则是用来保证程序在单线程中执行结果的正确性，但无法保证程序在多线程中执行的正确性。

第二条规则也比较容易理解，也就是说无论在单线程中还是多线程中，同一个锁如果处于被锁定的状态，那么必须先对锁进行了释放操作，后面才能继续进行lock操作。

第三条规则是一条比较重要的规则。直观地解释就是，如果一个线程先去写一个变量，然后一个线程去进行读取，那么写入操作肯定会先行发生于读操作。

第四条规则实际上就是体现happens-before原则具备传递性。

五、深入理解volatile关键字

1.volatile保证可见性

一旦一个共享变量（类的成员变量、类的静态成员变量）被volatile修饰之后，那么就具备了两层语义：

1）保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的。

2）禁止进行指令重排序。

先看一段代码，假如线程1先执行，线程2后执行：

```
//线程1
boolean stop = false;
while(!stop){
    doSomething();
}

//线程2
stop = true;
```

这段代码是很典型的一段代码，很多人在中断线程时可能都会采用这种标记办法。但是事实上，这段代码会完全运行正确么？即一定会将线程中断么？不一定，也许在大多数时候，这个代码能够把线程中断，但是也有可能会导致无法中断线程（虽然这个可能性很小，但是只要一旦发生这种情况就会造成死循环了）。

下面解释一下这段代码为何有可能导致无法中断线程。在前面已经解释过，每个线程在运行过程中都有自己的工作内存，那么线程1在运行的时候，会将stop变量的值拷贝一份放在自己的工作内存当中。

那么当线程**2**更改了**stop**变量的值之后，但是还没来得及写入主存当中，线程**2**转去做其他事情了，那么线程**1**由于不知道线程**2**对**stop**变量的更改，因此还会一直循环下去。

但是用volatile修饰之后就变得不一样了：

第一：使用volatile关键字会强制将修改的值立即写入主存；

第二：使用volatile关键字的话，当线程2进行修改时，会导致线程**1**的工作内存中缓存变量**stop**的缓存行无效（反映到硬件层的话，就是CPU的L1或者L2缓存中对应的缓存行无效）；

第三：由于线程1的工作内存中缓存变量stop的缓存行无效，所以线程**1**再次读取变量**stop**的值时会去主存读取。

那么在线程2修改stop值时（当然这里包括2个操作，修改线程2工作内存中的值，然后将修改后的值写入内存），会使得线程1的工作内存中缓存变量stop的缓存行无效，然后线程1读取时，发现自己的缓存行无效，它会等待缓存行对应的主存地址被更新之后，然后去对应的主存读取最新的值。

那么线程1读取到的就是最新的正确的值。

2.volatile不能确保原子性

下面看一个例子：

```
public class Nothing {

    private volatile int inc = 0;
    private volatile static int count = 10;

    private void increase() {
        ++inc;
    }

    public static void main(String[] args) {
        int loop = 10;
        Nothing nothing = new Nothing();
        while (loop-- > 0) {
            nothing.operation();
        }
    }

    private void operation() {
        final Nothing test = new Nothing();
        for (int i = 0; i < 10; i++) {
            new Thread(() -> {
                for (int j = 0; j < 1000000; j++) {
                    test.increase();
                }
                --count;
            }).start();
        }

        // 保证前面的线程都执行完
        while (count > 0) {

        }
        System.out.println("最后的数据为：" + test.inc);
    }

}
```

运行结果为：

```
最后的数据为：5919956
最后的数据为：3637231
最后的数据为：2144549
最后的数据为：2403538
最后的数据为：1762639
最后的数据为：2878721
最后的数据为：2658645
最后的数据为：2534078
最后的数据为：2031751
最后的数据为：2924506
```

大家想一下这段程序的输出结果是多少？也许有些朋友认为是1000000。但是事实上运行它会发现每次运行结果都不一致，都是一个小于1000000的数字。

可能有的朋友就会有疑问，不对啊，上面是对变量inc进行自增操作，由于volatile保证了可见性，那么在每个线程中对inc自增完之后，在其他线程中都能看到修改后的值啊，所以有10个线程分别进行了1000000次操作，那么最终inc的值应该是1000000*10=10000000。

这里面就有一个误区了，**volatile**关键字能保证可见性没有错，但是上面的程序错在没能保证原子性。 可见性只能保证每次读取的是最新的值，但是volatile没办法保证对变量的操作的原子性。

在前面已经提到过，自增操作是不具备原子性的，它包括读取变量的原始值、进行加**1**操作、写入工作内存。那么就是说自增操作的三个子操作可能会分割开执行，就有可能导致下面这种情况出现：

假如某个时刻变量inc的值为10，线程**1**对变量进行自增操作，线程**1**先读取了变量**inc**的原始值，然后线程**1**被阻塞了；

然后线程2对变量进行自增操作，线程2也去读取变量inc的原始值，由于线程**1**只是对变量**inc**进行读取操作，而没有对变量进行修改操作，所以不会导致线程**2**的工作内存中缓存变量**inc**的缓存行无效，也不会导致主存中的值刷新， 所以线程2会直接去主存读取inc的值，发现inc的值时10，然后进行加1操作，并把11写入工作内存，最后写入主存。

然后线程1接着进行加1操作，由于已经读取了inc的值，注意此时在线程1的工作内存中inc的值仍然为10，所以线程1对inc进行加1操作后inc的值为11，然后将11写入工作内存，最后写入主存。

那么两个线程分别进行了一次自增操作后，inc只增加了1。

根源就在这里，自增操作不是原子性操作，而且**volatile**也无法保证对变量的任何操作都是原子性的。

解决方案：可以通过**synchronized**或**lock**，进行加锁，来保证操作的原子性。也可以通过**AtomicInteger**。

在java 1.5的java.util.concurrent.atomic包下提供了一些原子操作类，即对基本数据类型的 自增（加1操作），自减（减1操作）、以及加法操作（加一个数），减法操作（减一个数）进行了封装，保证这些操作是原子性操作。**atomic**是利用**CAS**来实现原子性操作的（**Compare And Swap**），CAS实际上是利用处理器提供的**CMPXCHG**指令实现的，而处理器执行**CMPXCHG**指令是一个原子性操作。

3.volatile保证有序性

在前面提到volatile关键字能禁止指令重排序，所以volatile能在一定程度上保证有序性。

volatile关键字禁止指令重排序有两层意思：

1）当程序执行到volatile变量的读操作或者写操作时，在其前面的操作的更改肯定全部已经进行，且结果已经对后面的操作可见；在其后面的操作肯定还没有进行；

2）在进行指令优化时，不能将在对**volatile**变量的读操作或者写操作的语句放在其后面执行，也不能把**volatile**变量后面的语句放到其前面执行。

可能上面说的比较绕，举个简单的例子：

```
// x、y为非volatile变量
// flag为volatile变量

x = 2;         //语句1
y = 0;         //语句2
flag = true;   //语句3
x = 4;         //语句4
y = -1;        //语句5
```

由于**flag**变量为**volatile**变量，那么在进行指令重排序的过程的时候，不会将语句**3**放到语句**1**、语句**2**前面，也不会讲语句**3**放到语句**4**、语句**5**后面。但是要注意语句**1**和语句**2**的顺序、语句**4**和语句**5**的顺序是不作任何保证的。

并且volatile关键字能保证，执行到语句**3**时，语句**1**和语句**2**必定是执行完毕了的，且语句**1**和语句**2**的执行结果对语句**3**、语句**4**、语句**5**是可见的。

那么我们回到前面举的一个例子：

```
//线程1:
context = loadContext();   //语句1
inited = true;             //语句2

//线程2:
while(!inited ){
  sleep()
}
doSomethingwithconfig(context);
```

前面举这个例子的时候，提到有可能语句2会在语句1之前执行，那么久可能导致context还没被初始化，而线程2中就使用未初始化的context去进行操作，导致程序出错。

这里如果用volatile关键字对inited变量进行修饰，就不会出现这种问题了，因为当执行到语句**2**时，必定能保证**context**已经初始化完毕。

六、volatile的实现原理

1.可见性

处理器为了提高处理速度，不直接和内存进行通讯，而是将系统内存的数据独到内部缓存后再进行操作，但操作完后不知什么时候会写到内存。

如果对声明了**volatile**变量进行写操作时，**JVM**会向处理器发送一条**Lock**前缀的指令，将这个变量所在缓存行的数据写会到系统内存。 这一步确保了如果有其他线程对声明了volatile变量进行修改，则立即更新主内存中数据。

但这时候其他处理器的缓存还是旧的，所以在多处理器环境下，为了保证各个处理器缓存一致，每个处理会通过嗅探在总线上传播的数据来检查 自己的缓存是否过期， 当处理器发现自己缓存行对应的内存地址被修改了，就会将当前处理器的缓存行设置成无效状态，当处理器要对这个数据进行修改操作时，会强制重新从系统内存把数据读到处理器缓存里。 这一步确保了其他线程获得的声明了volatile变量都是从主内存中获取最新的。

2.有序性

Lock前缀指令实际上相当于一个内存屏障（也成内存栅栏），它确保指令重排序时不会把其后面的指令排到内存屏障之前的位置，也不会把前面的指令排到内存屏障的后面；即在执行到内存屏障这句指令时，在它前面的操作已经全部完成。

七、volatile的应用场景

synchronized关键字是防止多个线程同时执行一段代码，那么就会很影响程序执行效率，而volatile关键字在某些情况下性能要优于synchronized，但是要注意volatile关键字是无法替代synchronized关键字的，因为volatile关键字无法保证操作的原子性。通常来说，使用volatile必须具备以下2个条件：

1）对变量的写操作不依赖于当前值

2）该变量没有包含在具有其他变量的不变式中

下面列举几个Java中使用volatile的几个场景。

①.状态标记量

```
volatile boolean flag = false;
 //线程1
while(!flag){
    doSomething();
}
  //线程2
public void setFlag() {
    flag = true;
}
```

根据状态标记，终止线程。

②.单例模式中的double check

```
class Singleton {
    private volatile static Singleton instance = null;

    private Singleton() {

    }

    public static Singleton getInstance() {
        if (instance == null) {
            synchronized (Singleton.class) {
                if (instance == null)
                    instance = new Singleton();
            }
        }
        return instance;
    }
}
```

为什么要使用volatile 修饰instance？

主要在于instance = new Singleton()这句，这并非是一个原子操作，事实上在 JVM 中这句话大概做了下面 3 件事情:

1.给 instance 分配内存

2.调用 Singleton 的构造函数来初始化成员变量

3.将instance对象指向分配的内存空间（执行完这步 instance 就为非 null 了）。

但是在 JVM 的即时编译器中存在指令重排序的优化。也就是说上面的第二步和第三步的顺序是不能保证的，最终的执行顺序可能是 1-2-3 也可能是 1-3-2。如果是后者，则在 3 执行完毕、2 未执行之前，被线程二抢占了，这时 instance 已经是非 null 了（但却没有初始化），所以线程二会直接返回 instance，然后使用，然后顺理成章地报错。

时间： 2018-05-12 01:44:07

乐观锁与悲观锁

我们都知道，cpu是时分复用的，也就是把cpu的时间片，分配给不同的thread/process轮流执行，时间片与时间片之间，需要进行cpu切换，也就是会发生进程的切换。切换涉及到清空寄存器，缓存数据。然后重新加载新的thread所需数据。当一个线程被挂起时，加入到阻塞队列，在一定的时间或条件下，在通过notify()，notifyAll()唤醒回来。 在某个资源不可用的时候，就将**cpu**让出，把当前等待线程切换为阻塞状态。等到资源**(**比如一个共享数据）可用了，那么就将线程唤醒，让他进入**runnable**状态等待**cpu**调度。这就是典型的悲观锁的实现。 独占锁是一种悲观锁，synchronized就是一种独占锁，它假设最坏的情况，认为一个线程修改共享数据的时候其他线程也会修改该数据，因此只在确保其它线程不会造成干扰的情况下执行，会导致其它所有需要锁的线程挂起，等待持有锁的线程释放锁。

但是，由于在进程挂起和恢复执行过程中存在着很大的开销。当一个线程正在等待锁时，它不能做任何事，所以悲观锁有很大的缺点。举个例子，如果一个线程需要某个资源，但是这个资源的占用时间很短，当线程第一次抢占这个资源时，可能这个资源被占用，如果此时挂起这个线程，可能立刻就发现资源可用，然后又需要花费很长的时间重新抢占锁，时间代价就会非常的高。

所以就有了乐观锁的概念，他的核心思路就是，每次不加锁而是假设修改数据之前其他线程一定不会修改，如果因为修改过产生冲突就失败就重试，直到成功为止。在上面的例子中，某个线程可以不让出cpu，而是一直while循环，如果失败就重试，直到成功为止。所以，当数据争用不严重时，乐观锁效果更好。比如CAS就是一种乐观锁思想的应用。

CAS

CAS 操作包含三个操作数 —— 内存位置（**V**）、预期原值（**A**）和新值**(B)**。执行**CAS**操作的时候，将内存位置的值与预期原值比较，如果相匹配，那么处理器会自动将该位置值更新为新值。否则，处理器不做任何操作。

举个CAS操作的应用场景的一个例子，当一个线程需要修改共享变量的值。完成这个操作，先取出共享变量的值赋给A，然后基于A的基础进行计算，得到新值B，完了需要更新共享变量的值了，这个时候就可以调用CAS方法更新变量值了。

在java中可以通过锁和循环CAS的方式来实现原子操作。Java中 `java.util.concurrent.atomic` 包相关类就是 CAS的实现，atomic包里包括以下类：

类名	说明
AtomicBoolean	可以用原子方式更新的 `boolean` 值。
AtomicInteger	可以用原子方式更新的 `int` 值。
AtomicIntegerArray	可以用原子方式更新其元素的 `int` 数组。
AtomicIntegerFieldUpdater	基于反射的实用工具，可以对指定类的指定 `volatile int` 字段进行原子更新。
AtomicLong	可以用原子方式更新的 `long` 值。
AtomicLongArray	可以用原子方式更新其元素的 `long` 数组。
AtomicLongFieldUpdater	基于反射的实用工具，可以对指定类的指定 `volatile long` 字段进行原子更新。
AtomicMarkableReference	`AtomicMarkableReference` 维护带有标记位的对象引用，可以原子方式对其进行更新。
AtomicReference	可以用原子方式更新的对象引用。
AtomicReferenceArray	可以用原子方式更新其元素的对象引用数组。
AtomicReferenceFieldUpdater	基于反射的实用工具，可以对指定类的指定 `volatile` 字段进行原子更新。
AtomicStampedReference	`AtomicStampedReference` 维护带有整数“标志”的对象引用，可以用原子方式对其进行更新。

下面我们来已AtomicIneger的源码为例来看看CAS操作：

```
public final int getAndAdd(int delta) {
    for (; ; ) {
        int current = get();
        int next = current + delta;
        if (compareAndSet(current, next))
            return current;
    }
}
```

这里很显然使用CAS操作（for(;;)里面），他每次都从内存中读取数据，+1操作，然后两个值进行CAS操作。如果成功则返回，否则失败重试，直到修改成功为止。上面源码最关键的地方有两个，一个for循环，它代表着一种宁死不屈的精神，不成功誓不罢休。还有就是compareAndSet：

```
public final boolean compareAndSet(int expect, int update) {
    return unsafe.compareAndSwapInt(this, valueOffset, expect, u
pdate);
}
```

compareAndSet方法内部是调用Java本地方法compareAndSwapInt来实现的，而compareAndSwapInt方法内部又是借助C来调用CPU的底层指令来保证在硬件层面上实现原子操作的。在intel处理器中，CAS是通过调用**cmpxchg**指令完成的。这就是我们常说的**CAS**操作（compare and swap）。

CAS的问题

CAS虽然很高效的解决原子操作，但是CAS仍然存在三大问题。ABA问题，循环时间长开销大和只能保证一个共享变量的原子操作。

1. ABA问题。因为CAS需要在操作值的时候检查下值有没有发生变化，如果没有发生变化则更新，但是如果一个值原来是A，变成了B，又变成了A，那么使用CAS进行检查时会发现它的值没有发生变化，但是实际上却变化了。ABA问题的解决思路就是使用版本号。在变量前面追加上版本号，每次变量更新的时候把版本号加一，那么A－B－A 就会变成1A-2B－3A。从Java1.5开始JDK的atomic包里提供了一个类AtomicStampedReference来解决ABA问题。这个类的compareAndSet方法作用是首先检查当前引用是否等于预期引用，并且当前

标志是否等于预期标志，如果全部相等，则以原子方式将该引用和该标志的值设置为给定的更新值。

2. 循环时间长开销大。自旋CAS如果长时间不成功，会给CPU带来非常大的执行开销。
3. 只能保证一个共享变量的原子操作。当对一个共享变量执行操作时，我们可以使用循环CAS的方式来保证原子操作，但是对多个共享变量操作时，循环CAS就无法保证操作的原子性，这个时候就可以用锁，或者有一个取巧的办法，就是把多个共享变量合并成一个共享变量来操作。比如有两个共享变量i＝2,j=a，合并一下ij=2a，然后用CAS来操作ij。从Java1.5开始JDK提供了AtomicReference类来保证引用对象之间的原子性，你可以把多个变量放在一个对象里来进行CAS操作。

一、AQS简介

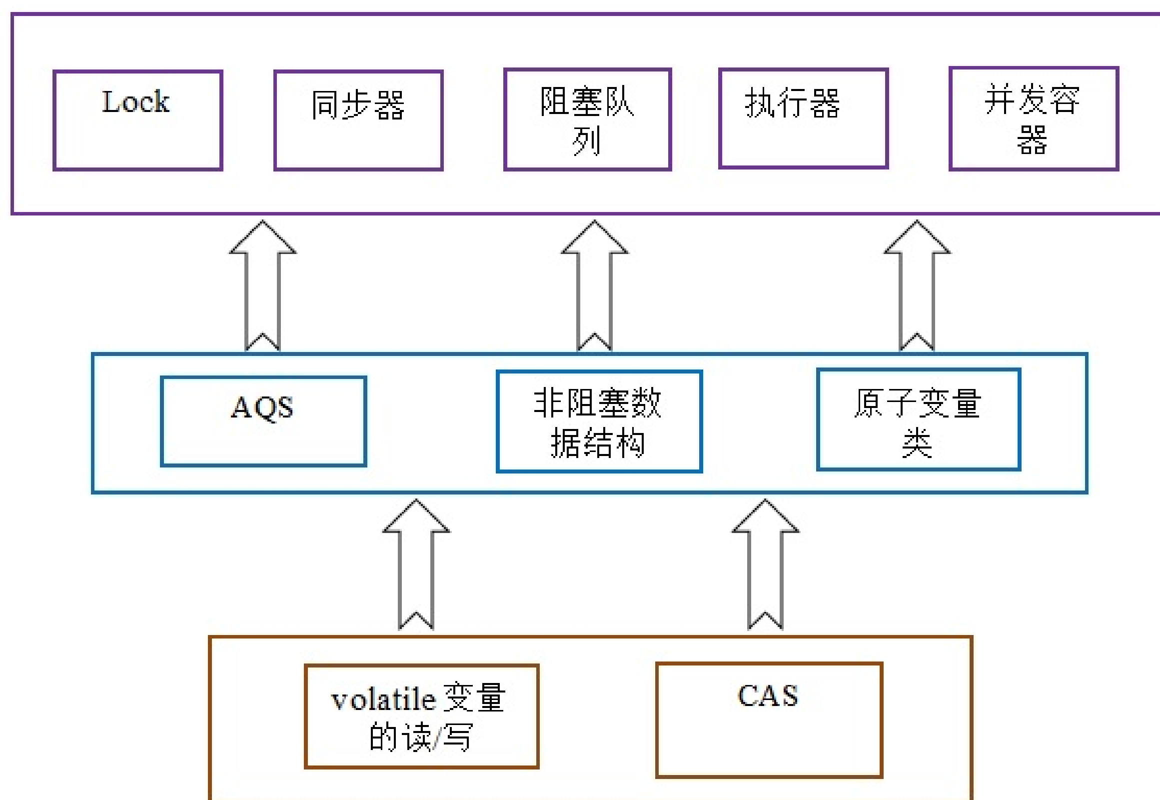
1.1 AQS介绍

AbstractQueuedSynchronizer提供了一个基于FIFO队列，可以用于构建锁或者其他相关同步装置的基础框架。该同步器（以下简称同步器）利用了一个int来表示状态，期望它能够成为实现大部分同步需求的基础。使用的方法是继承，子类通过继承同步器并需要实现它的方法来管理其状态，管理的方式就是通过类似acquire和release的方式来操纵状态。然而多线程环境中对状态的操纵必须确保原子性，因此子类对于状态的把握，需要使用这个同步器提供的以下三个方法对状态进行操作：

```
java.util.concurrent.locks.AbstractQueuedSynchronizer.getState()
java.util.concurrent.locks.AbstractQueuedSynchronizer.setState(int)
java.util.concurrent.locks.AbstractQueuedSynchronizer.compareAndSetState(int, int)
```

子类推荐被定义为自定义同步装置的内部类，同步器自身没有实现任何同步接口，它仅仅是定义了若干acquire之类的方法来供使用。该同步器即可以作为排他模式也可以作为共享模式，当它被定义为一个排他模式时，其他线程对其的获取就被阻止，而共享模式对于多个线程获取都可以成功。

1.2 AQS用处



1.3 同步器与锁

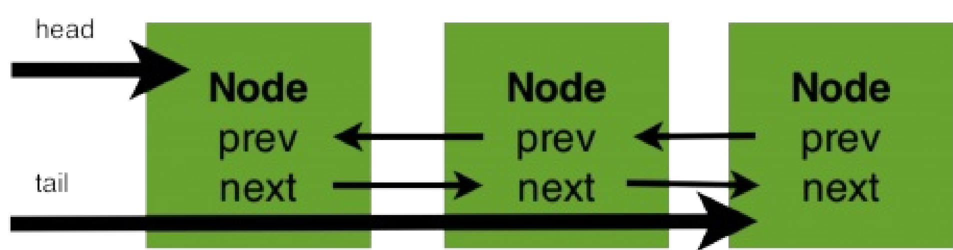
同步器是实现锁的关键，利用同步器将锁的语义实现，然后在锁的实现中聚合同步器。 可以这样理解：锁的API是面向使用者的，它定义了与锁交互的公共行为，而每个锁需要完成特定的操作也是透过这些行为来完成的（比如：可以允许两个线程进行加锁，排除两个以上的线程），但是实现是依托给同步器来完成；同步器面向的是线程访问和资源控制，它定义了线程对资源是否能够获取以及线程的排队等操作。锁和同步器很好的隔离了二者所需要关注的领域，严格意义上讲，同步器可以适用于除了锁以外的其他同步设施上（包括锁）。 同步器的开始提到了其实现依赖于一个FIFO队列，那么队列中的元素Node就是保存着线程引用和线程状态的容器，每个线程对同步器的访问，都可以看做是队列中的一个节点。Node的主要包含以下成员变量：

```
Node {
    int waitStatus;
    Node prev;
    Node next;
    Node nextWaiter;
    Thread thread;
}
```

以上五个成员变量主要负责保存该节点的线程引用，同步等待队列（以下简称sync队列）的前驱和后继节点，同时也包括了同步状态。

属性名称	描述
int waitStatus	表示节点的状态。其中包含的状态有： CANCELLED，值为1，表示当前的线程被取消；SIGNAL，值为-1，表示当前节点的后继节点包含的线程需要运行，也就是unpark；CONDITION，值为-2，表示当前节点在等待condition，也就是在condition队列中；PROPAGATE，值为-3，表示当前场景下后续的acquireShared能够得以执行；值为0，表示当前节点在sync队列中，等待着获取锁。
Node prev	前驱节点，比如当前节点被取消，那就需要前驱节点和后继节点来完成连接。
Node next	后继节点。
Node nextWaiter	存储condition队列中的后继节点。
Thread thread	入队列时的当前线程。

节点成为sync队列和condition队列构建的基础，在同步器中就包含了sync队列。同步器拥有三个成员变量：sync队列的头结点head、sync队列的尾节点tail和状态state。对于锁的获取，请求形成节点，将其挂载在尾部，而锁资源的转移（释放再获取）是从头部开始向后进行。对于同步器维护的状态state，多个线程对其的获取将会产生一个链式的结构。



1.4 API说明

实现自定义同步器时，需要使用同步器提供的getState()、setState()和compareAndSetState()方法来操纵状态的变迁。

方法名称	描述
protected boolean tryAcquire(int arg)	排它的获取这个状态。这个方法的实现需要查询当前状态是否允许获取，然后再进行获取（使用compareAndSetState来做）状态。
protected boolean tryRelease(int arg)	释放状态。
protected int tryAcquireShared(int arg)	共享的模式下获取状态。
protected boolean tryReleaseShared(int arg)	共享的模式下释放状态。
protected boolean isHeldExclusively()	在排它模式下，状态是否被占用。

实现这些方法必须是非阻塞而且是线程安全的，推荐使用该同步器的父类java.util.concurrent.locks.AbstractOwnableSynchronizer来设置当前的线程。 开始提到同步器内部基于一个FIFO队列，对于一个独占锁的获取和释放有以下伪码可以表示。 获取一个排他锁。

```
while(获取锁) {
    if (获取到) {
        退出while循环
    } else {
        if(当前线程没有入队列) {
            那么入队列
        }
        阻塞当前线程
    }
}
释放一个排他锁。
```

```
if (释放成功) {
    删除头结点
    激活原头结点的后继节点
}
```

1.5 Mutex 示例

下面通过一个排它锁的例子来深入理解一下同步器的工作原理，而只有掌握同步器的工作原理才能够更加深入了解其他的并发组件。排他锁的实现，一次只能一个线程获取到锁。

```
class Mutex implements Lock, java.io.Serializable {
   // 内部类，自定义同步器
   private static class Sync extends AbstractQueuedSynchronizer
{
     // 是否处于占用状态
     protected boolean isHeldExclusively() {
       return getState() == 1;
     }
     // 当状态为0的时候获取锁
     public boolean tryAcquire(int acquires) {
       assert acquires == 1; // Otherwise unused
       if (compareAndSetState(0, 1)) {
         setExclusiveOwnerThread(Thread.currentThread());
         return true;
       }
       return false;
     }
     // 释放锁，将状态设置为0
     protected boolean tryRelease(int releases) {
       assert releases == 1; // Otherwise unused
       if (getState() == 0) throw new IllegalMonitorStateExcepti
on();
       setExclusiveOwnerThread(null);
       setState(0);
       return true;
     }
     // 返回一个Condition，每个condition都包含了一个condition队列
```

```
        Condition newCondition() { return new ConditionObject(); }
    }
    // 仅需要将操作代理到Sync上即可
    private final Sync sync = new Sync();
    public void lock()                { sync.acquire(1); }
    public boolean tryLock()          { return sync.tryAcquire(1)
; }
    public void unlock()              { sync.release(1); }
    public Condition newCondition()   { return sync.newCondition(
); }
    public boolean isLocked()         { return sync.isHeldExclusi
vely(); }
    public boolean hasQueuedThreads() { return sync.hasQueuedThre
ads(); }
    public void lockInterruptibly() throws InterruptedException {
        sync.acquireInterruptibly(1);
    }
    public boolean tryLock(long timeout, TimeUnit unit)
            throws InterruptedException {
        return sync.tryAcquireNanos(1, unit.toNanos(timeout));
    }
 }
```

可以看到Mutex将Lock接口均代理给了同步器的实现。使用方将Mutex构造出来之后，调用lock获取锁，调用unlock进行解锁。下面以Mutex为例子，详细分析以下同步器的实现逻辑。

二、独占模式

2.1 acquire

实现分析 public final void acquire(int arg) 该方法以排他的方式获取锁，对中断不敏感，完成synchronized语义。

```
public final void acquire(int arg) {
        if (!tryAcquire(arg) &&
            acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
            selfInterrupt();
}
```

上述逻辑主要包括：

1. 尝试获取（调用tryAcquire更改状态，需要保证原子性）；在tryAcquire方法中使用了同步器提供的对state操作的方法，利用compareAndSet保证只有一个线程能够对状态进行成功修改，而没有成功修改的线程将进入sync队列排队。
2. 如果获取不到，将当前线程构造成节点Node并加入sync队列；进入队列的每个线程都是一个节点Node，从而形成了一个双向队列，类似CLH队列，这样做的目的是线程间的通信会被限制在较小规模（也就是两个节点左右）。
3. 再次尝试获取，如果没有获取到那么将当前线程从线程调度器上摘下，进入等待状态。使用LockSupport将当前线程unpark，关于LockSupport后续会详细介绍。

```
private Node addWaiter(Node mode) {
    Node node = new Node(Thread.currentThread(), mode);
    // 快速尝试在尾部添加
    Node pred = tail;
    if (pred != null) {
        node.prev = pred;
        if (compareAndSetTail(pred, node)) {
            pred.next = node;
            return node;
        }
    }
    enq(node);
    return node;
}

private Node enq(final Node node) {
    for (;;) {
        Node t = tail;
        if (t == null) { // Must initialize
            if (compareAndSetHead(new Node()))
                tail = head;
        } else {
            node.prev = t;
            if (compareAndSetTail(t, node)) {
            t.next = node;
            return t;
        }
    }
}
```

上述逻辑主要包括：

1. 使用当前线程构造Node；对于一个节点需要做的是将当节点前驱节点指向尾节点（current.prev = tail），尾节点指向它（tail = current），原有的尾节点的后继节点指向它（t.next = current）而这些操作要求是原子的。上面的操作是利用尾节点的设置来保证的，也就是compareAndSetTail来完成的。

2. 先行尝试在队尾添加；

如果尾节点已经有了，然后做如下操作：

i. 分配引用T指向尾节点；
ii. 将节点的前驱节点更新为尾节点（current.prev = tail）；
iii. 如果尾节点是T，那么将当尾节点设置为该节点（tail = current，原子更新）；
iv. T的后继节点指向当前节点（T.next = current）。 注意第3点是要求原子的。 这样可以以最短路径O(1)的效果来完成线程入队，是最大化减少开销的一种方式。

3. 如果队尾添加失败或者是第一个入队的节点。

如果是第1个节点，也就是sync队列没有初始化，那么会进入到enq这个方法，进入的线程可能有多个，或者说在addWaiter中没有成功入队的线程都将进入enq这个方法。

可以看到enq的逻辑是确保进入的Node都会有机会顺序的添加到sync队列中，而加入的步骤如下：

i. 如果尾节点为空，那么原子化的分配一个头节点，并将尾节点指向头节点，这一步是初始化；
ii. 然后是重复在**addWaiter**中做的工作，但是在一个**for(;;)**的循环中，直到当前节点入队为止。

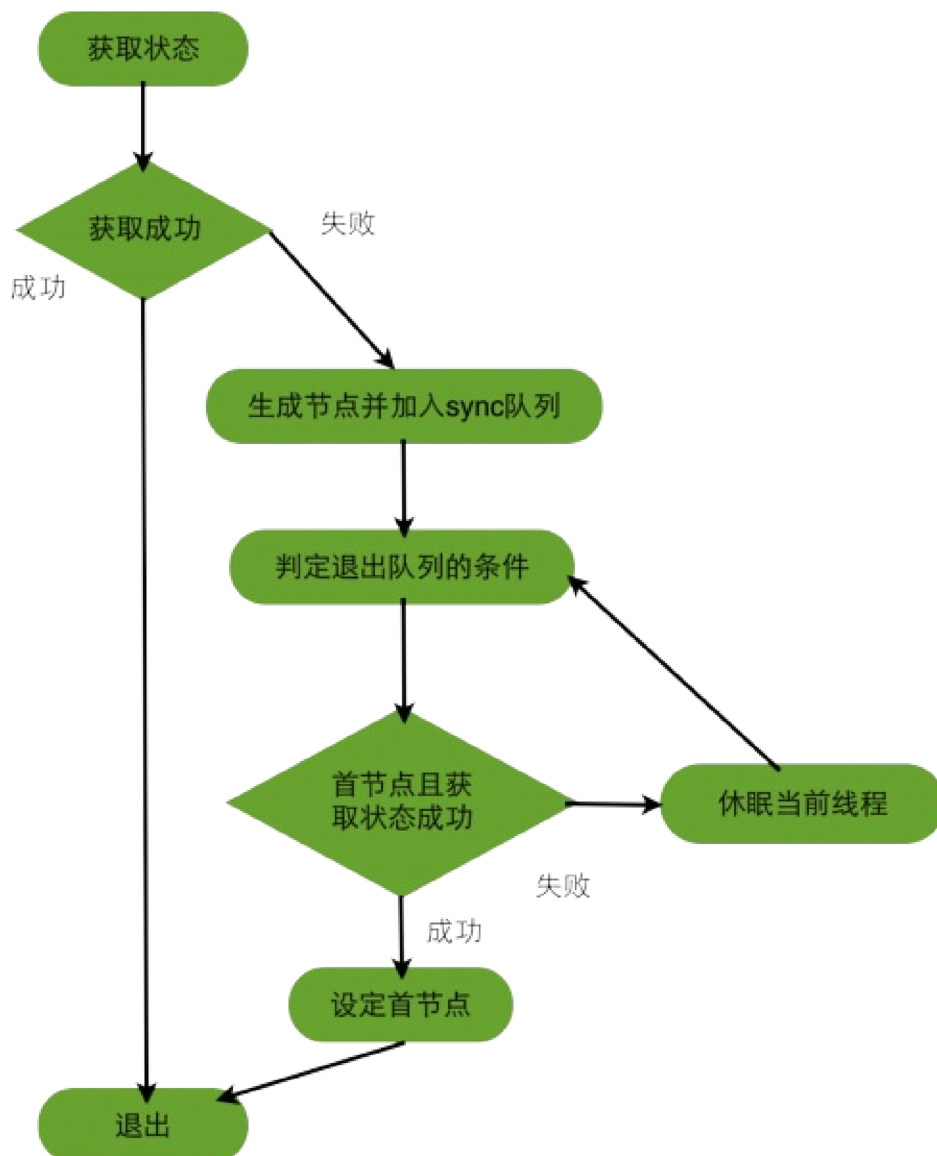
进入sync队列之后，接下来就是要进行锁的获取，或者说是访问控制了，只有一个线程能够在同一时刻继续的运行，而其他的进入等待状态。而每个线程都是一个独立的个体，它们自省的观察，当条件满足的时候（自己的前驱是头结点并且原子性的获取了状态），那么这个线程能够继续运行。

```
final boolean acquireQueued(final Node node, int arg) {
    boolean failed = true;
    try {
        boolean interrupted = false;
        for (;;) {
            final Node p = node.predecessor();
            if (p == head &&tryAcquire(arg)) {
                setHead(node);
                p.next = null; // help GC
                failed = false;
                return interrupted;
            }
            if (shouldParkAfterFailedAcquire(p, node) &&
                    parkAndCheckInterrupt())
                interrupted = true;
                }
    } finally {
        if (failed)
            cancelAcquire(node);
    }
}
```

上述逻辑主要包括：

1. 获取当前节点的前驱节点； 需要获取当前节点的前驱节点，而头结点所对应的含义是当前占有锁且正在运行。
2. 当前驱节点是头结点并且能够获取状态，代表该当前节点占有锁； 如果满足上述条件，那么代表能够占有锁，根据节点对锁占有的含义，设置头结点为当前节点。
3. 否则进入等待状态。 如果没有轮到当前节点运行，那么将当前线程从线程调度器上摘下，也就是进入等待状态。 这里针对acquire做一下总结：
4. 状态的维护； 需要在锁定时，需要维护一个状态(int类型)，而对状态的操作是原子和非阻塞的，通过同步器提供的对状态访问的方法对状态进行操纵，并且利用compareAndSet来确保原子性的修改。
5. 状态的获取； 一旦成功的修改了状态，当前线程或者说节点，就被设置为头节点。
6. sync队列的维护。 在获取资源未果的过程中条件不符合的情况下(不该自己，前驱节点不是头节点或者没有获取到资源)进入睡眠状态，停止线程调度器对当

前节点线程的调度。这时引入的一个释放的问题，也就是说使睡眠中的Node或者说线程获得通知的关键，就是前驱节点的通知，而这一个过程就是释放，释放会通知它的后继节点从睡眠中返回准备运行。下面的流程图基本描述了一次acquire所需要经历的过程：



如上图所示，其中的判定退出队列的条件，判定条件是否满足和休眠当前线程就是完成了自旋spin的过程。

2.2 release

public final boolean release(int arg) 在unlock方法的实现中，使用了同步器的release方法。相对于在之前的acquire方法中可以得出调用acquire，保证能够获取到锁（成功获取状态），而release则表示将状态设置回去，也就是将资源释放，或者说将锁释放。

```
public final boolean release(int arg) {
    if (tryRelease(arg)) {
        Node h = head;
        if (h != null && h.waitStatus != 0)
            unparkSuccessor(h);
        return true;
    }
    return false;
}
```

上述逻辑主要包括：

1. 尝试释放状态；tryRelease能够保证原子化的将状态设置回去，当然需要使用compareAndSet来保证。如果释放状态成功过之后，将会进入后继节点的唤醒过程。
2. 唤醒当前节点的后继节点所包含的线程。通过LockSupport的unpark方法将休眠中的线程唤醒，让其继续acquire状态。

```
private void unparkSuccessor(Node node) {
    // 将状态设置为同步状态
    int ws = node.waitStatus;
    if (ws < 0)
        compareAndSetWaitStatus(node, ws, 0);

    /*
    * 获取当前节点的后继节点，如果满足状态，那么进行唤醒操作
    * 如果没有满足状态，从尾部开始找寻符合要求的节点并将其唤醒
    */

    Node s = node.next;
    if (s == null || s.waitStatus > 0) {
        s = null;
        for (Node t = tail; t != null && t != node; t = t.prev)
            if (t.waitStatus <= 0)
                s = t;
        }
    if (s != null)
        LockSupport.unpark(s.thread);
}
```

上述逻辑主要包括，该方法取出了当前节点的next引用，然后对其线程(Node)进行了唤醒，这时就只有一个或合理个数的线程被唤醒，被唤醒的线程继续进行对资源的获取与争夺。回顾整个资源的获取和释放过程：在获取时，维护了一个sync队列，每个节点都是一个线程在进行自旋，而依据就是自己是否是首节点的后继并且能够获取资源；在释放时，仅仅需要将资源还回去，然后通知一下后继节点并将其唤醒。这里需要注意，队列的维护（首节点的更换）是依靠消费者（获取时）来完成的，也就是说在满足了自旋退出的条件时的一刻，这个节点就会被设置成为首节点。

2.3 tryAcquire

protected boolean tryAcquire(int arg) tryAcquire是自定义同步器需要实现的方法，也就是自定义同步器非阻塞原子化的获取状态，如果锁该方法一般用于Lock的tryLock实现中，这个特性是synchronized无法提供的。

public final void acquireInterruptibly(int arg) 该方法提供获取状态能力，当然在无法获取状态的情况下会进入sync队列进行排队，这类似acquire，但是和acquire不同的地方在于它能够在外界对当前线程进行中断的时候提前结束获取状态的操作，换句话说，就是在类似synchronized获取锁时，外界能够对当前线程进行中断，并且获取锁的这个操作能够响应中断并提前返回。一个线程处于synchronized块中或者进行同步I/O操作时，对该线程进行中断操作，这时该线程的中断标识位被设置为true，但是线程依旧继续运行。 如果在获取一个通过网络交互实现的锁时，这个锁资源突然进行了销毁，那么使用acquireInterruptibly的获取方式就能够让该时刻尝试获取锁的线程提前返回。而同步器的这个特性被实现Lock接口中的lockInterruptibly方法。根据Lock的语义，在被中断时，lockInterruptibly将会抛出InterruptedException来告知使用者。

```
public final void acquireInterruptibly(int arg)
    throws InterruptedException {
    if (Thread.interrupted())
        throw new InterruptedException();
    if (!tryAcquire(arg))
        doAcquireInterruptibly(arg);
}

private void doAcquireInterruptibly(int arg)
    throws InterruptedException {
    final Node node = addWaiter(Node.EXCLUSIVE);
    boolean failed = true;
    try {
        for (;;) {
            final Node p = node.predecessor();
            if (p == head && tryAcquire(arg)) {
                setHead(node);
                p.next = null; // help GC
                failed = false;
                return;
            }
            // 检测中断标志位
            if (shouldParkAfterFailedAcquire(p, node) &&
            parkAndCheckInterrupt())
                throw new InterruptedException();
        }
    } finally {
        if (failed)
            cancelAcquire(node);
    }
}
```

上述逻辑主要包括：

1. 检测当前线程是否被中断；判断当前线程的中断标志位，如果已经被中断了，那么直接抛出异常并将中断标志位设置为false。
2. 尝试获取状态；调用tryAcquire获取状态，如果顺利会获取成功并返回。
3. 构造节点并加入sync队列；获取状态失败后，将当前线程引用构造为节点并加入到sync队列中。退出队列的方式在没有中断的场景下和acquireQueued类

似，当头结点是自己的前驱节点并且能够获取到状态时，即可以运行，当然要将本节点设置为头结点，表示正在运行。
4. 中断检测。 在每次被唤醒时，进行中断检测，如果发现当前线程被中断，那么抛出InterruptedException并退出循环。

2.4 doAcquireNanos

private boolean doAcquireNanos(int arg, long nanosTimeout) throws InterruptedException 该方法提供了具备有超时功能的获取状态的调用，如果在指定的nanosTimeout内没有获取到状态，那么返回false，反之返回true。可以将该方法看做acquireInterruptibly的升级版，也就是在判断是否被中断的基础上增加了超时控制。 针对超时控制这部分的实现，主要需要计算出睡眠的delta，也就是间隔值。间隔可以表示为nanosTimeout = 原有nanosTimeout – now（当前时间）+ lastTime（睡眠之前记录的时间）。如果nanosTimeout大于0，那么还需要使当前线程睡眠，反之则返回false。

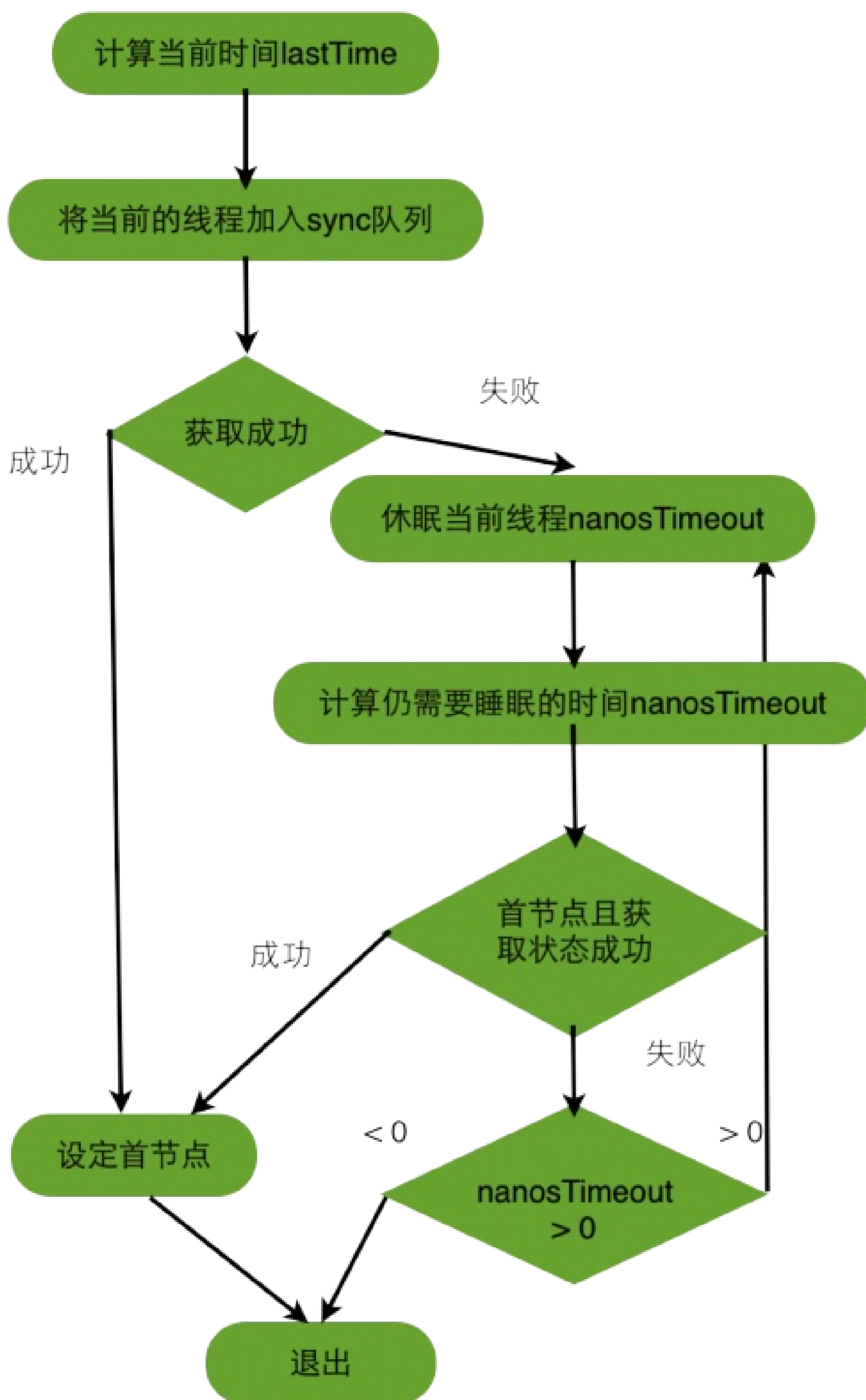
```
private boolean doAcquireNanos(int arg, long nanosTimeout)
throws InterruptedException {
    long lastTime = System.nanoTime();
    final Node node = addWaiter(Node.EXCLUSIVE);
    boolean failed = true;
    try {
        for (;;) {
            final Node p = node.predecessor();
            if (p == head &&tryAcquire(arg)) {
                setHead(node);
                p.next = null; // help GC
                failed = false;
                return true;
            }
            if (nanosTimeout <= 0)                  return false;
             if (shouldParkAfterFailedAcquire(p, node) && nanosT
imeout > spinForTimeoutThreshold)
            LockSupport.parkNanos(this, nanosTimeout);
            long now = System.nanoTime();
            //计算时间，当前时间减去睡眠之前的时间得到睡眠的时间，然后被
            //原有超时时间减去，得到了还应该睡眠的时间
            nanosTimeout -= now - lastTime;
            lastTime = now;
            if (Thread.interrupted())
                throw new InterruptedException();
        }
    } finally {
        if (failed)
            cancelAcquire(node);
    }
}
```

上述逻辑主要包括：

1. 加入sync队列；将当前线程构造成为节点Node加入到sync队列中。
2. 条件满足直接返回；退出条件判断，如果前驱节点是头结点并且成功获取到状态，那么设置自己为头结点并退出，返回true，也就是在指定的nanosTimeout之前获取了锁。
3. 获取状态失败休眠一段时间；通过LockSupport.unpark来指定当前线程休眠一

段时间。

4. 计算再次休眠的时间；唤醒后的线程，计算仍需要休眠的时间，该时间表示为nanosTimeout = 原有nanosTimeout – now（当前时间）+ lastTime（睡眠之前记录的时间）。其中now – lastTime表示这次睡眠所持续的时间。
5. 休眠时间的判定。唤醒后的线程，计算仍需要休眠的时间，并无阻塞的尝试再获取状态，如果失败后查看其nanosTimeout是否大于0，如果小于0，那么返回完全超时，没有获取到锁。如果nanosTimeout小于等于1000L纳秒，则进入快速的自旋过程。那么快速自旋会造成处理器资源紧张吗？结果是不会，经过测算，开销看起来很小，几乎微乎其微。Doug Lea应该测算了在线程调度器上的切换造成的额外开销，因此在短时1000纳秒内就让当前线程进入快速自旋状态，如果这时再休眠相反会让nanosTimeout的获取时间变得更加不精确。上述过程可以如下图所示：

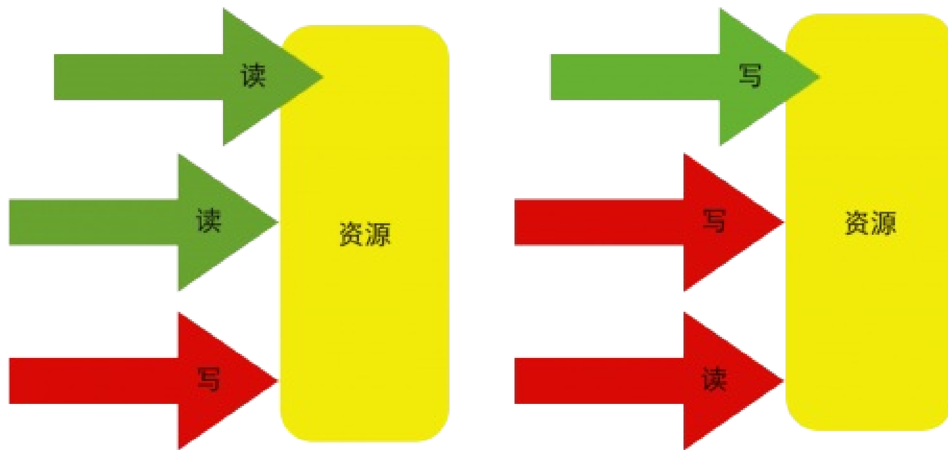


上述这个图中可以理解为在类似获取状态需要排队的基础上增加了一个超时控制的逻辑。每次超时的时间就是当前超时剩余的时间减去睡眠的时间，而在这个超时时间的基础上进行了判断，如果大于0那么继续睡眠（等待），可以看出这个超时版本的获取状态只是一个近似超时的获取状态，因此任何含有超时的调用基本结果就是近似于给定超时。

三、共享模式

3.1 acquireShared

public final void acquireShared(int arg) 调用该方法能够以共享模式获取状态，共享模式和之前的独占模式有所区别。以文件的查看为例，如果一个程序在对其进行读取操作，那么这一时刻，对这个文件的写操作就被阻塞，相反，这一时刻另一个程序对其进行同样的读操作是可以进行的。如果一个程序在对其进行写操作，那么所有的读与写操作在这一时刻就被阻塞，直到这个程序完成写操作。 以读写场景为例，描述共享和独占的访问模式，如下图所示：



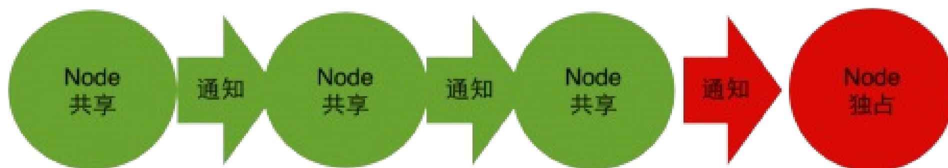
上图中，红色代表被阻塞，绿色代表可以通过。

```
public final void acquireShared(int arg) {
    if (tryAcquireShared(arg) < 0)
      doAcquireShared(arg);
}
private void doAcquireShared(int arg) {
  final Node node = addWaiter(Node.SHARED);
  boolean failed = true;
  try {
    boolean interrupted = false;
    for (;;) {
      final Node p = node.predecessor();
      if (p == head) {
        int r = tryAcquireShared(arg);
        if (r >= 0) {
          setHeadAndPropagate(node, r);
          p.next = null; // help GC
          if (interrupted)
            selfInterrupt();
            failed = false;
            return;
                }
            }
      if (shouldParkAfterFailedAcquire(p, node) &&
parkAndCheckInterrupt())
            interrupted = true;
    }
  } finally {
    if (failed)
    cancelAcquire(node);
    }
}
```

上述逻辑主要包括：

1. 尝试获取共享状态；调用tryAcquireShared来获取共享状态，该方法是非阻塞的，如果获取成功则立刻返回，也就表示获取共享锁成功。
2. 获取失败进入sync队列；在获取共享状态失败后，当前时刻有可能是独占锁被其他线程所把持，那么将当前线程构造成为节点（共享模式）加入到sync队列中。

3. 循环内判断退出队列条件；如果当前节点的前驱节点是头结点并且获取共享状态成功，这里和独占锁acquire的退出队列条件类似。
4. 获取共享状态成功；在退出队列的条件上，和独占锁之间的主要区别在于获取共享状态成功之后的行为，而如果共享状态获取成功之后会判断后继节点是否是共享模式，如果是共享模式，那么就直接对其进行唤醒操作，也就是同时激发多个线程并发的运行。
5. 获取共享状态失败。通过使用LockSupport将当前线程从线程调度器上摘下，进入休眠状态。对于上述逻辑中，节点之间的通知过程如下图所示：



上图中，绿色表示共享节点，它们之间的通知和唤醒操作是在前驱节点获取状态时就进行的，红色表示独占节点，它的被唤醒必须取决于前驱节点的释放，也就是release操作，可以看出来图中的独占节点如果要运行，必须等待前面的共享节点均释放了状态才可以。而独占节点如果获取了状态，那么后续的独占式获取和共享式获取均被阻塞。

3.2 releaseShared

public final boolean releaseShared(int arg) 调用该方法释放共享状态，每次获取共享状态acquireShared都会操作状态，同样在共享锁释放的时候，也需要将状态释放。比如说，一个限定一定数量访问的同步工具，每次获取都是共享的，但是如果超过了一定的数量，将会阻塞后续的获取操作，只有当之前获取的消费者将状态释放才可以使阻塞的获取操作得以运行。

```
public final boolean releaseShared(int arg) {
    if (tryReleaseShared(arg)) {
        doReleaseShared();
        return true;
    }
    return false;
}
```

上述逻辑主要就是调用同步器的tryReleaseShared方法来释放状态，并同时在doReleaseShared方法中唤醒其后继节点。

3.3 一个例子 TwinsLock

在上述对同步器AbstractQueuedSynchronizer进行了实现层面的分析之后，我们通过一个例子来加深对同步器的理解：设计一个同步工具，该工具在同一时刻，只能有两个线程能够并行访问，超过限制的其他线程进入阻塞状态。对于这个需求，可以利用同步器完成一个这样的设定，定义一个初始状态，为2，一个线程进行获取那么减1，一个线程释放那么加1，状态正确的范围在[0，1，2]三个之间，当在0时，代表再有新的线程对资源进行获取时只能进入阻塞状态（注意在任何时候进行状态变更的时候均需要以CAS作为原子性保障）。由于资源的数量多于1个，同时可以有两个线程占有资源，因此需要实现tryAcquireShared和tryReleaseShared方法，这里谢谢luoyuyou和同事小明指正，已经修改了实现。

```
public class TwinsLock implements Lock {
    private final Sync    sync    = new Sync(2);

    private static final class Sync extends AbstractQueuedSynchr
onizer {
        private static final long    serialVersionUID    = -7889
272986162341211L;

        Sync(int count) {
            if (count <= 0) {
                throw new IllegalArgumentException("count must l
arge than zero.");
            }
            setState(count);
        }

        public int tryAcquireShared(int reduceCount) {
            for (;;) {
                int current = getState();
                int newCount = current - reduceCount;
                if (newCount < 0 || compareAndSetState(current,
newCount)) {
                    return newCount;
                }
            }
        }
```

```
        public boolean tryReleaseShared(int returnCount) {
            for (;;) {
                int current = getState();
                int newCount = current + returnCount;
                if (compareAndSetState(current, newCount)) {
                    return true;
                }
            }
        }
    }

    public void lock() {
        sync.acquireShared(1);
    }

    public void lockInterruptibly() throws InterruptedException
{
        sync.acquireSharedInterruptibly(1);
    }

    public boolean tryLock() {
        return sync.tryAcquireShared(1) >= 0;
    }

    public boolean tryLock(long time, TimeUnit unit) throws Inte
rruptedException {
        return sync.tryAcquireSharedNanos(1, unit.toNanos(time))
;
    }

    public void unlock() {
        sync.releaseShared(1);
    }

    @Override
    public Condition newCondition() {
        return null;
    }
}
```

上述测试用例的逻辑主要包括：

1. 打印线程 Worker在两次睡眠之间打印自身线程，如果一个时刻只能有两个线程同时访问，那么打印出来的内容将是成对出现。
2. 分隔线程 不停的打印换行，能让Worker的输出看起来更加直观。 该测试的结果是在一个时刻，仅有两个线程能够获得到锁，并完成打印，而表象就是打印的内容成对出现。

四、总结

AQS简核心是通过一个共享变量来同步状态，变量的状态由子类去维护，而AQS框架做的是：

- 线程阻塞队列的维护
- 线程阻塞和唤醒

共享变量的修改都是通过Unsafe类提供的CAS操作完成的。
AbstractQueuedSynchronizer类的主要方法是acquire和release，典型的模板方法， 下面这4个方法由子类去实现：

```
protected boolean tryAcquire(int arg)
protected boolean tryRelease(int arg)
protected int tryAcquireShared(int arg)
protected boolean tryReleaseShared(int arg)
```

acquire方法用来获取锁，返回true说明线程获取成功继续执行，一旦返回false则线程加入到等待队列中，等待被唤醒，release方法用来释放锁。 一般来说实现的时候这两个方法被封装为lock和unlock方法。

前言

java5之后，并发包中新增了Lock接口（以及相关实现类）用来实现锁的功能，它提供了与synchronized关键字类似的同步功能。既然有了synchronized这种内置的锁功能，为何要新增Lock接口？先来想象一个场景：手把手的进行锁获取和释放，先获得锁A，然后再获取锁B，当获取锁B后释放锁A同时获取锁C，当锁C获取后，再释放锁B同时获取锁D，以此类推，这种场景下，synchronized关键字就不那么容易实现了，而使用Lock却显得容易许多。

定义

```
public class ReentrantLock implements Lock, java.io.Serializable
{
    private final Sync sync;
    abstract static class Sync extends AbstractQueuedSynchronizer
 {

        /**
         * Performs {@link Lock#lock}. The main reason for subcl
assing
         * is to allow fast path for nonfair version.
         */
        abstract void lock();

        /**
         * Performs non-fair tryLock.  tryAcquire is implemented
 in
         * subclasses, but both need nonfair try for trylock met
hod.
         */
        final boolean nonfairTryAcquire(int acquires) {
            final Thread current = Thread.currentThread();
            int c = getState();
            if (c == 0) {
                if (compareAndSetState(0, acquires)) {
                    setExclusiveOwnerThread(current);
```

```
                return true;
            }
        }
        else if (current == getExclusiveOwnerThread()) {
            int nextc = c + acquires;
            if (nextc < 0) // overflow
                throw new Error("Maximum lock count exceeded"
);
            setState(nextc);
            return true;
        }
        return false;
    }

    protected final boolean tryRelease(int releases) {
        int c = getState() - releases;
        if (Thread.currentThread() != getExclusiveOwnerThrea
d())
            throw new IllegalMonitorStateException();
        boolean free = false;
        if (c == 0) {
            free = true;
            setExclusiveOwnerThread(null);
        }
        setState(c);
        return free;
    }

}
//默认非公平锁
public ReentrantLock() {
    sync = new NonfairSync();
}
//fair为false时，采用公平锁策略
public ReentrantLock(boolean fair) {
    sync = fair ? new FairSync() : new NonfairSync();
}
public void lock() {
    sync.lock();
}
```

```
    public void unlock() {    sync.release(1);}
    public Condition newCondition() {
        return sync.newCondition();
    }
    ...
}
```

从源代码可以Doug lea巧妙的采用组合模式把lock和unlock方法委托给同步器完成。

使用方式

```
Lock lock = new ReentrantLock();
Condition condition = lock.newCondition();
lock.lock();
try {
  while(条件判断表达式) {
      condition.wait();
  }
 // 处理逻辑
} finally {
    lock.unlock();
}
```

需要显示的获取锁，并在finally块中显示的释放锁，目的是保证在获取到锁之后，最终能够被释放。

非公平锁实现

在非公平锁中，每当线程执行lock方法时，都尝试利用CAS把state从0设置为1。

那么Doug lea是如何实现锁的非公平性呢？ 我们假设这样一个场景：

1. 持有锁的线程A正在running，队列中有线程BCDEF被挂起并等待被唤醒；
2. 在某一个时间点，线程A执行unlock，唤醒线程B；
3. 同时线程G执行lock，这个时候会发生什么？线程B和G拥有相同的优先级，这

里讲的优先级是指获取锁的优先级，同时执行CAS指令竞争锁。如果恰好线程G成功了，线程B就得重新挂起等待被唤醒。

通过上述场景描述，我们可以看书，即使线程B等了很长时间也得和新来的线程G同时竞争锁，如此的不公平。

```
static final class NonfairSync extends Sync {
    /**
     * Performs lock.  Try immediate barge, backing up to normal
     * acquire on failure.
     */
    final void lock() {
        if (compareAndSetState(0, 1))
            setExclusiveOwnerThread(Thread.currentThread());
        else
            acquire(1);
    }

    public final void acquire(int arg) {
        if (!tryAcquire(arg) &&
                acquireQueued(addWaiter(Node.EXCLUSIVE), arg))

          selfInterrupt();
    }

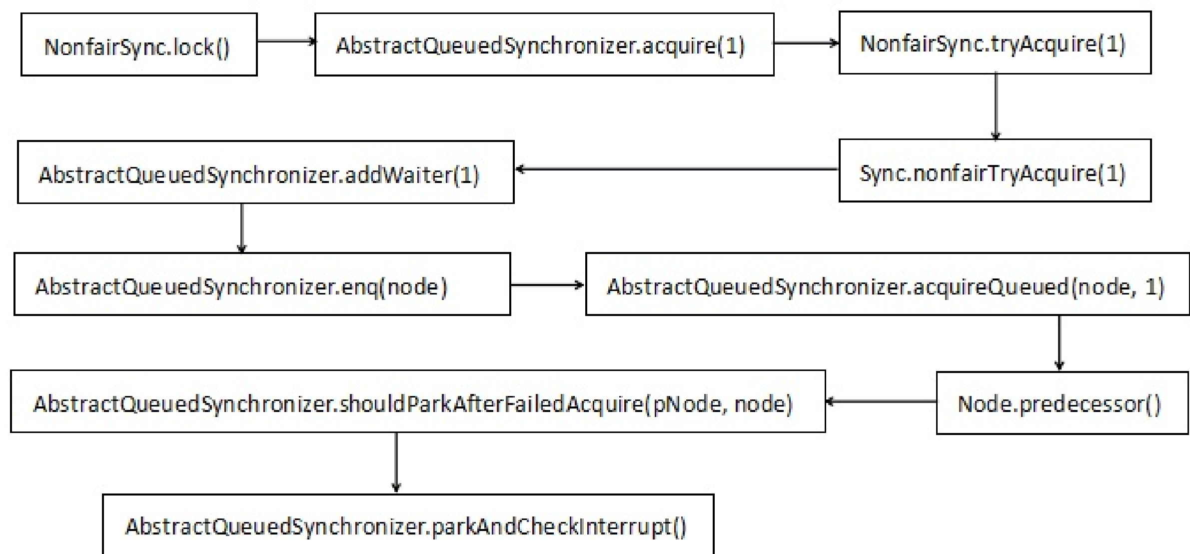
    protected final boolean tryAcquire(int acquires) {
        return nonfairTryAcquire(acquires);
    }
}
```

下面我们用线程A和线程B来描述非公平锁的竞争过程。

1. 线程A和B同时执行CAS指令，假设线程A成功，线程B失败，则表明线程A成功获取锁，并把同步器中的exclusiveOwnerThread设置为线程A。
2. 竞争失败的线程B，在nonfairTryAcquire方法中，会再次尝试获取锁，

   Doug lea会在多处尝试重新获取锁，应该是在这段时间如果线程A释放锁，线程B就可以直接获取锁而不用挂起

- 完整的执行流程如下：



公平锁实现

在公平锁中，每当线程执行lock方法时，如果同步器的队列中有线程在等待，则直接加入到队列中。 场景分析：

1. 持有锁的线程A正在running，对列中有线程BCDEF被挂起并等待被唤醒；
2. 线程G执行lock，队列中有线程BCDEF在等待，线程G直接加入到队列的对尾。

所以每个线程获取锁的过程是公平的，等待时间最长的会最先被唤醒获取锁。

```
static final class FairSync extends Sync {
    private static final long serialVersionUID = -30008978970904
66540L;

    final void lock() {
        acquire(1);
    }

    /**
     * Fair version of tryAcquire.  Don't grant access unless
     * recursive call or no waiters or is first.
     */
    protected final boolean tryAcquire(int acquires) {
        final Thread current = Thread.currentThread();
        int c = getState();
        if (c == 0) {
            if (!hasQueuedPredecessors() &&
                compareAndSetState(0, acquires)) {
                setExclusiveOwnerThread(current);
                return true;
            }
        }
        else if (current == getExclusiveOwnerThread()) {
            int nextc = c + acquires;
            if (nextc < 0)
                throw new Error("Maximum lock count exceeded");
            setState(nextc);
            return true;
        }
        return false;
    }
}
```

重入锁实现

重入锁，即线程可以重复获取已经持有的锁。在非公平和公平锁中，都对重入锁进行了实现。

```
    if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0)
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
    }
```

条件变量Condition

条件变量很大一个程度上是为了解决Object.wait/notify/notifyAll难以使用的问题。

```
public class ConditionObject implements Condition, java.io.Seria
lizable {
    /** First node of condition queue. */
    private transient Node firstWaiter;
    /** Last node of condition queue. */
    private transient Node lastWaiter;
    public final void signal() {}
    public final void signalAll() {}
    public final void awaitUninterruptibly() {}
    public final void await() throws InterruptedException {}
}
```

1. Synchronized中，所有的线程都在同一个object的条件队列上等待。而ReentrantLock中，每个condition都维护了一个条件队列。
2. 每一个**Lock**可以有任意数据的**Condition**对象，**Condition**是与**Lock**绑定的，所以就有**Lock**的公平性特性：如果是公平锁，线程为按照FIFO的顺序从*Condition.await*中释放，如果是非公平锁，那么后续的锁竞争就不保证FIFO顺序了。
3. Condition接口定义的方法，**await**对应于**Object.wait**，**signal**对应于**Object.notify**，**signalAll**对应于**Object.notifyAll**。特别说明的是Condition的接口改变名称就是为了避免与Object中的*wait/notify/notifyAll*的语义和使用上混淆。

先看一个condition在生产者消费者的应用场景：

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

/**
 * Created by j_zhan on 2016/7/13.
 */
public class Queue<T> {
    private final T[] items;
    private final Lock lock = new ReentrantLock();
    private Condition notFull = lock.newCondition();
    private Condition notEmpty = lock.newCondition();
    private int head, tail, count;
    public Queue(int maxSize) {
        items = (T[]) new Object[maxSize];
    }
    public Queue() {
        this(10);
    }

    public void put(T t) throws InterruptedException {
        lock.lock();
        try {
            while (count == items.length) {
                //数组满时，线程进入等待队列挂起。线程被唤醒时，从这里返
回。
                notFull.await();
            }
            items[tail] = t;
            if (++tail == items.length) {
                tail = 0;
            }
            ++count;
            notEmpty.signal();
        } finally {
            lock.unlock();
        }
    }

    public T take() throws InterruptedException {
```

```
            lock.lock();
            try {
                while (count == 0) {
                    notEmpty.await();
                }
                T o = items[head];
                items[head] = null;//GC
                if (++head == items.length) {
                    head = 0;
                }
                --count;
                notFull.signal();
                return o;
            } finally {
                lock.unlock();
            }
        }
}
```

假设线程AB在并发的往items中插入数据，当items中元素存满时。如果线程A获取到锁，继续添加数据，满足count == items.length条件，导致线程A执行await方法。 **ReentrantLock**是独占锁，同一时刻只有一个线程能获取到锁，所以在lock.lock()和lock.unlock()之间可能有一次释放锁的操作（同样也必然还有一次获取锁的操作）。在Queue类中，不管take还是put，在线程持有锁之后只有await()方法有可能释放锁，然后挂起线程，一旦条件满足就被唤醒，再次获取锁。具体实现如下：

```
public final void await() throws InterruptedException {
    if (Thread.interrupted())
        throw new InterruptedException();
    Node node = addConditionWaiter();
    int savedState = fullyRelease(node);
    int interruptMode = 0;
    while (!isOnSyncQueue(node)) {
        LockSupport.park(this);
        if ((interruptMode = checkInterruptWhileWaiting(node)) !
= 0)
            break;
    }
    if (acquireQueued(node, savedState) && interruptMode != THRO
W_IE)
        interruptMode = REINTERRUPT;
    if (node.nextWaiter != null) // clean up if cancelled
        unlinkCancelledWaiters();
    if (interruptMode != 0)
        reportInterruptAfterWait(interruptMode);
}

private Node addConditionWaiter() {
    Node t = lastWaiter;
    // If lastWaiter is cancelled, clean out.
    if (t != null && t.waitStatus != Node.CONDITION) {
        unlinkCancelledWaiters();
        t = lastWaiter;
    }
    Node node = new Node(Thread.currentThread(), Node.CONDITION)
;
    if (t == null)
        firstWaiter = node;
    else
        t.nextWaiter = node;
    lastWaiter = node;
    return node;
}
```

await实现逻辑：

1. 将线程A加入到条件等待队列中，如果最后一个节点是取消状态，则从对列中删除。
2. 线程A释放锁，实质上是线程A修改AQS的状态state为0，并唤醒AQS等待队列中的线程B，线程B被唤醒后，尝试获取锁，接下去的过程就不重复说明了。
3. 线程A释放锁并唤醒线程B之后，如果线程A不在AQS的同步队列中，线程A将通过LockSupport.park进行挂起操作。
4. 随后，线程A等待被唤醒，当线程A被唤醒时，会通过acquireQueued方法竞争锁，如果失败，继续挂起。如果成功，线程A从await位置恢复。

假设线程B获取锁之后，执行了take操作和条件变量的signal，signal通过某种实现唤醒了线程A，具体实现如下：

```
public final void signal() {
    if (!isHeldExclusively())
        throw new IllegalMonitorStateException();
    Node first = firstWaiter;
    if (first != null)
        doSignal(first);
}

private void doSignal(Node first) {
    do {
        if ((firstWaiter = first.nextWaiter) == null)
            lastWaiter = null;
        first.nextWaiter = null;
    } while (!transferForSignal(first) &&
             (first = firstWaiter) != null);
}

final boolean transferForSignal(Node node) {
   if (!compareAndSetWaitStatus(node, Node.CONDITION, 0))
       return false;
   Node p = enq(node); //线程A插入到AQS的等待队列中
   int ws = p.waitStatus;
   if (ws > 0 || !compareAndSetWaitStatus(p, ws, Node.SIGNAL))
       LockSupport.unpark(node.thread);
   return true;
}
```

signal实现逻辑：

1. 接着上述场景，线程B执行了signal方法，取出条件队列中的第一个非CANCELLED节点线程，即线程A。另外，signalAll就是唤醒条件队列中所有非CANCELLED节点线程。遇到CANCELLED线程就需要将其从队列中删除。
2. 通过CAS修改线程A的waitStatus，表示该节点已经不是等待条件状态，并将线程A插入到AQS的等待队列中。
3. 唤醒线程A，线程A和别的线程进行锁的竞争。

总结

1. ReentrantLock提供了内置锁类似的功能和内存语义。
2. 此外，ReetrantLock还提供了其它功能，包括定时的锁等待、可中断的锁等待、公平性、以及实现非块结构的加锁、Condition，对线程的等待和唤醒等操作更加灵活，一个ReentrantLock可以有多个Condition实例，所以更有扩展性，不过ReetrantLock需要显示的获取锁，并在finally中释放锁，否则后果很严重。
3. ReentrantLock在性能上似乎优于Synchronized，其中在jdk1.6中略有胜出，在1.5中是远远胜出。那么为什么不放弃内置锁，并在新代码中都使用ReetrantLock？
4. 在java1.5中， 内置锁与ReentrantLock相比有例外一个优点：在线程转储中能给出在哪些调用帧中获得了哪些锁，并能够检测和识别发生死锁的线程。Reentrant的非块状特性任然意味着，获取锁的操作不能与特定的栈帧关联起来，而内置锁却可以。
5. 因为内置锁时JVM的内置属性，所以未来更可能提升synchronized而不是ReentrantLock的性能。例如对线程封闭的锁对象消除优化，通过增加锁粒度来消除内置锁的同步。

前言

本文的主要详细分析ArrayBlockingQueue的实现原理，由于该并发集合其底层是使用了java.util.ReentrantLock和java.util.Condition来完成并发控制的，我们可以通过JDK的源代码更好的学习这些并发控制类的使用，同时该类也是所有并发集合中最简单的一个，分析该类的源码也是为之后分析其他并发集合做好基础。

一、Queue接口和BlockingQueue接口回顾

1.1 Queue接口回顾

> 在Queue接口中，除了继承Collection接口中定义的方法外，它还分别额外地定义插入、删除、查询这3个操作，其中每一个操作都以两种不同的形式存在，每一种形式都对应着一个方法。

方法说明：

	Throws exception	Returns special value
Insert	add(e)	offer(e)
Remove	remove()	poll()
Examine	element()	peek()

1. add方法在将一个元素插入到队列的尾部时，如果出现队列已经满了，那么就会抛出IllegalStateException,而使用offer方法时，如果队列满了，则添加失败,返回false,但并不会引发异常。
2. remove方法是获取队列的头部元素并且删除，如果当队列为空时，那么就会抛出NoSuchElementException。而poll在队列为空时，则返回一个null。
3. element方法是从队列中获取到队列的第一个元素，但不会删除，但是如果队列为空时，那么它就会抛出NoSuchElementException。peek方法与之类似，只是不会抛出异常，而是返回false。

后面我们在分析ArrayBlockingQueue的方法时，主要也是围绕着这几个方法来进行分析。

1.2 BlockingQueue接口回顾

> BlockingQueue是JDK1.5出现的接口，它在原来的Queue接口基础上提供了更多的额外功能：当获取队列中的头部元素时，如果队列为空，那么它将会使执行线程处于等待状态；当添加一个元素到队列的尾部时，如果队列已经满了，那么它同样会使执行的线程处于等待状态。

前面我们在说Queue接口时提到过，它针对于相同的操作提供了2种不同的形式，而BlockingQueue更夸张，针对于相同的操作提供了4种不同的形式。

> 该四种形式分别为：
>
> 1. 抛出异常
> 2. 返回一个特殊值(可能是null或者是false，取决于具体的操作)
> 3. 阻塞当前执行直到其可以继续
> 4. 当线程被挂起后，等待最大的时间，如果一旦超时，即使该操作依旧无法继续执行，线程也不会再继续等待下去。

对应的方法说明:

	Throws exception	Returns special value	Blocks	Times out
Insert	add(e)	offer(e)	put(e)	offer(e, time, unit)
Remove	remove()	poll()	take()	poll(time, unit)
Examine	element()	peek()	无	无

BlockingQueue虽然比起Queue在操作上提供了更多的支持，但是它在使用有如下的几点:

> 1. BlockingQueue中是不允许添加null的，该接受在声明的时候就要求所有的实现类在接收到一个null的时候，都应该抛出NullPointerException。
> 2. BlockingQueue是线程安全的，因此它的所有和队列相关的方法都具有原子性。但是对于那么从Collection接口中继承而来的批量操作方法，比如addAll(Collection e)等方法，BlockingQueue的实现通常没有保证其具有原子性，因此我们在使用的BlockingQueue，应该尽可能地不去使用这些方法。
> 3. BlockingQueue主要应用于生产者与消费者的模型中，其元素的添加和获取都是极具规律性的。但是对于remove(Object o)这样的方法，虽然BlockingQueue可以保证元素正确的删除，但是这样的操作会非常响应性能，因此我们在没有特殊的情况下，也应该避免使用这类方法。

二、ArrayBlockingQueue深入分析

有了上面的铺垫，下面我们就可以真正开始分析ArrayBlockingQueue了。在分析之前，首先让我们看看API对其的描述。 **注意:这里使用的JDK版本为1.7，不同的JDK版本在实现上存在不同**

```
40 /**
41  * A bounded {@linkplain BlockingQueue blocking queue} backed by an
42  * array.  This queue orders elements FIFO (first-in-first-out).  The
43  * <em>head</em> of the queue is that element that has been on the
44  * queue the longest time.  The <em>tail</em> of the queue is that
45  * element that has been on the queue the shortest time. New elements
46  * are inserted at the tail of the queue, and the queue retrieval
47  * operations obtain elements at the head of the queue.
48  *
49  * <p>This is a classic &quot;bounded buffer&quot;, in which a
50  * fixed-sized array holds elements inserted by producers and
51  * extracted by consumers.  Once created, the capacity cannot be
52  * changed.  Attempts to {@code put} an element into a full queue
53  * will result in the operation blocking; attempts to {@code take} an
54  * element from an empty queue will similarly block.
55  *
56  * <p>This class supports an optional fairness policy for ordering
57  * waiting producer and consumer threads.  By default, this ordering
58  * is not guaranteed. However, a queue constructed with fairness set
59  * to {@code true} grants threads access in FIFO order. Fairness
60  * generally decreases throughput but reduces variability and avoids
61  * starvation.
```

ArrayBlocking是一个大小固定的BlockingQueue，并且其底层是由一个数组所维护。队列的元素顺序按照先进先出的规则。

新的元素总是插入到队列的尾部，获取的元素总是从队列的头部开始获取。

ArrayBlockingQueue一旦创建，其大小就不能再改变了。

当队列满的时候，执行put操作，将会使线程阻塞，当队列为空的时候，执行take操作，也将会使线程阻塞。

ArrayBlocking支持公平策略。(其底层是通过ReenetrantLock的公平锁来完成的)。默认情况下是使用的非公平锁。

使用公平锁会造成吞吐的下降但是可以避免线程饥饿。

首先让我们看下ArrayBlockingQueue的核心组成：

```
/** 底层维护队列元素的数组 */
    final Object[] items;

    /**  当读取元素时数组的下标(这里称为读下标) */
    int takeIndex;

    /** 添加元素时数组的下标 (这里称为写小标)*/
    int putIndex;

    /** 队列中的元素个数 */
    int count;

    /**用于并发控制的工具类**/
    final ReentrantLock lock;

    /** 控制take操作时是否让线程等待 */
    private final Condition notEmpty;

    /** 控制put操作时是否让线程等待 */
    private final Condition notFull;
```

> take方法分析(369-379行):

```
    public E take() throws InterruptedException {
        final ReentrantLock lock = this.lock;
        /*
            尝试获取锁，如果此时锁被其他线程锁占用，那么当前线程就处于Wait
ing的状态。
            注意:当方法是支持线程中断响应的如果其他线程此时中断当前线程，
            那么当前线程就会抛出InterruptedException
         */
        lock.lockInterruptibly();
        try {
            /*
          如果此时队列中的元素个数为0,那么就让当前线程wait,并且释放锁。
          注意:这里使用了while进行重复检查，是为了防止当前线程可能由于其他
未知的原因被唤醒。
          (通常这种情况被称为"spurious wakeup")
            */
        while (count == 0)
                notEmpty.await();
            //如果队列不为空，则从队列的头部取元素
            return extract();
        } finally {
             //完成锁的释放
            lock.unlock();
        }
    }
```

> extract方法分析(163-171):

```
    /*
      根据takeIndex来获取当前的元素,然后通知其他等待的线程。
      Call only when holding lock.(只有当前线程已经持有了锁之后，它才能调用该方法)
     */
    private E extract() {
        final Object[] items = this.items;

        //根据takeIndex获取元素,因为元素是一个Object类型的数组,因此它通过cast方法将其转换成泛型。
        E x = this.<E>cast(items[takeIndex]);

        //将当前位置的元素设置为null
        items[takeIndex] = null;

        //并且将takeIndex++,注意：这里因为已经使用了锁，因此inc方法中没有使用到原子操作
        takeIndex = inc(takeIndex);

        //将队列中的总的元素减1
        --count;
        //唤醒其他等待的线程
        notFull.signal();
        return x;
    }
```

> put方法分析(318-239)

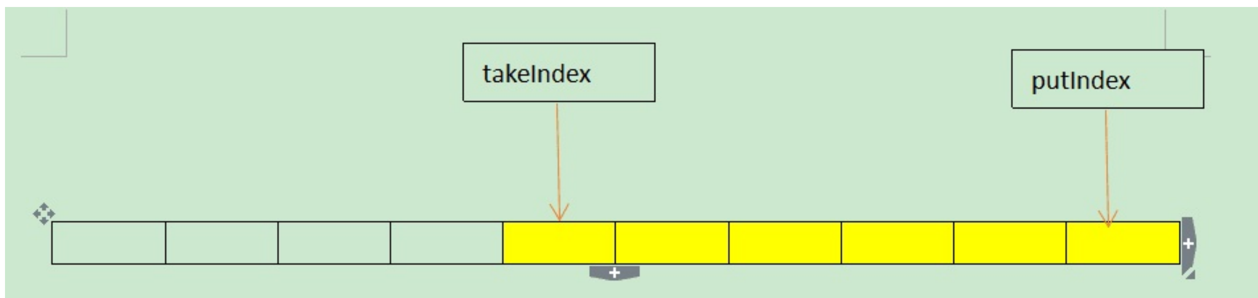
```
    public void put(E e) throws InterruptedException {
        //首先检查元素是否为空，否则抛出NullPointerException
        checkNotNull(e);
        final ReentrantLock lock = this.lock;
        //进行锁的抢占
        lock.lockInterruptibly();
        try {
            /*当队列的长度等于数组的长度,此时说明队列已经满了,这里同样
              使用了while来方式当前线程被"伪唤醒"。*/
            while (count == items.length)
                //则让当前线程处于等待状态
                notFull.await();
            //一旦获取到锁并且队列还未满时，则执行insert操作。
            insert(e);
        } finally {
            //完成锁的释放
            lock.unlock();
        }
    }

      //检查元素是否为空
     private static void checkNotNull(Object v) {
        if (v == null)
            throw new NullPointerException();
      }

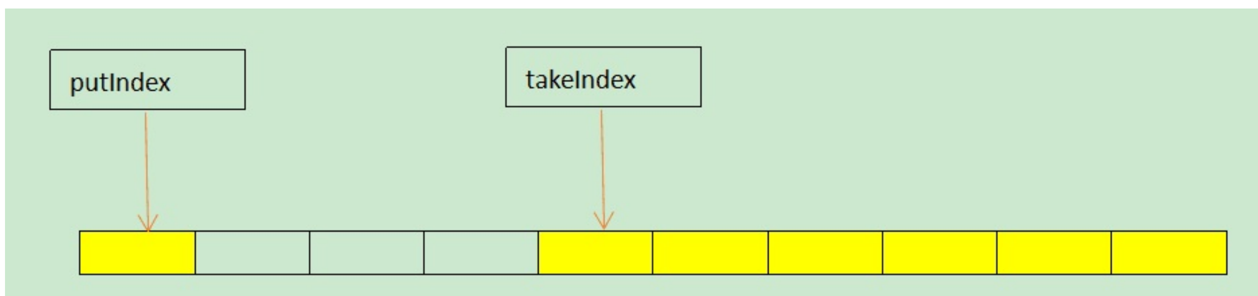
     //该方法的逻辑非常简单
    private void insert(E x) {
        //将当前元素设置到putIndex位置
        items[putIndex] = x;
        //让putIndex++
        putIndex = inc(putIndex);
        //将队列的大小加1
        ++count;
        //唤醒其他正在处于等待状态的线程
        notEmpty.signal();
    }
```

注:**ArrayBlockingQueue**其实是一个循环队列 我们使用一个图来简单说明一下:

> 黄色表示数组中有元素



当再一次执行put的时候,其结果为：



此时放入的元素会从头开始置，我们通过其incr方法更加清晰的看出其底层的操作：

```
    /**
     * Circularly increment i.
     */
    final int inc(int i) {
        //当takeIndex的值等于数组的长度时,就会重新置为0，这个一个循环递增的过程
        return (++i == items.length) ? 0 : i;
    }
```

至此，ArrayBlockingQueue的核心部分就分析完了，其余的队列操作基本上都是换汤不换药的，此处不再一一列举。

前言

在前面的文章ArrayBlockingQueue中，已经对JDK中的BlockingQueue中的做了一个回顾，同时对ArrayBlockingQueue中的核心方法作了说明，而LinkedBlockingQueue作为JDK中BlockingQueue家族系列中一员，由于其作为固定大小线程池(Executors.newFixedThreadPool())底层所使用的阻塞队列，分析它的目的主要在于2点：

1. 与ArrayBlockingQueue进行类比学习，加深各种数据结构的理解
2. 了解底层实现，能够更好地理解每一种阻塞队列对线程池性能的影响，做到真正的知其然，且知其所以然

3. 源码分析LinkedBlockingQueue的实现

4. 与ArrayBlockingQueue进行比较
5. 说明为什么选择LinkedBlockingQueue作为固定大小的线程池的阻塞队列

LinkedBlockingQueue深入分析

LinkedBlockingQueue，见名之意，它是由一个基于链表的阻塞队列，首先看一下的核心组成：

```
    // 所有的元素都通过Node这个静态内部类来进行存储，这与LinkedList的处理方式完全一样
    static class Node<E> {
        //使用item来保存元素本身
        E item;
        //保存当前节点的后继节点
        Node<E> next;
        Node(E x) { item = x; }
    }
    /**
        阻塞队列所能存储的最大容量
        用户可以在创建时手动指定最大容量,如果用户没有指定最大容量
        那么最默认的最大容量为Integer.MAX_VALUE.
    */
    private final int capacity;
```

```
    /**
        当前阻塞队列中的元素数量
        PS:如果你看过ArrayBlockingQueue的源码,你会发现
        ArrayBlockingQueue底层保存元素数量使用的是一个
        普通的int类型变量。其原因是在ArrayBlockingQueue底层
        对于元素的入队列和出队列使用的是同一个lock对象。而数
        量的修改都是在处于线程获取锁的情况下进行操作，因此不
        会有线程安全问题。
        而LinkedBlockingQueue却不是，它的入队列和出队列使用的是两个
        不同的lock对象,因此无论是在入队列还是出队列，都会涉及对元素数
        量的并发修改，(之后通过源码可以更加清楚地看到)因此这里使用了一个原
子操作类
        来解决对同一个变量进行并发修改的线程安全问题。
    */
    private final AtomicInteger count = new AtomicInteger(0);

    /**
     * 链表的头部
     * LinkedBlockingQueue的头部具有一个不变性:
     * 头部的元素总是为null，head.item==null
     */
    private transient Node<E> head;

    /**
     * 链表的尾部
     * LinkedBlockingQueue的尾部也具有一个不变性:
     * 即last.next==null
     */
    private transient Node<E> last;

    /**
     元素出队列时线程所获取的锁
     当执行take、poll等操作时线程需要获取的锁
    */
    private final ReentrantLock takeLock = new ReentrantLock();

    /**
    当队列为空时，通过该Condition让从队列中获取元素的线程处于等待状态
    */
```

```
    private final Condition notEmpty = takeLock.newCondition();

    /**
      元素入队列时线程所获取的锁
      当执行add、put、offer等操作时线程需要获取锁
    */
    private final ReentrantLock putLock = new ReentrantLock();

    /**
     当队列的元素已经达到capactiy，通过该Condition让元素入队列的线程处于
等待状态
    */
    private final Condition notFull = putLock.newCondition();
```

通过上面的分析，我们可以发现LinkedBlockingQueue在入队列和出队列时使用的不是同一个Lock，这也意味着它们之间的操作不会存在互斥操作。在多个CPU的情况下，它们可以做到真正的在同一时刻既消费、又生产，能够做到并行处理。

下面让我们看下LinkedBlockingQueue的构造方法：

```
    /**
     * 如果用户没有显示指定capacity的值，默认使用int的最大值
     */
    public LinkedBlockingQueue() {
        this(Integer.MAX_VALUE);
    }
    /**
     可以看到，当队列中没有任何元素的时候，此时队列的头部就等于队列的尾部，
     指向的是同一个节点，并且元素的内容为null
    */
    public LinkedBlockingQueue(int capacity) {
        if (capacity <= 0) throw new IllegalArgumentException();
        this.capacity = capacity;
        last = head = new Node<E>(null);
    }

    /*
    在初始化LinkedBlockingQueue的时候，还可以直接将一个集合
    中的元素全部入队列，此时队列最大容量依然是int的最大值。
```

```
    */
    public LinkedBlockingQueue(Collection<? extends E> c) {
        this(Integer.MAX_VALUE);
        final ReentrantLock putLock = this.putLock;
        //获取锁
        putLock.lock(); // Never contended, but necessary for vi
sibility
        try {
            //迭代集合中的每一个元素,让其入队列,并且更新一下当前队列中的元
素数量
            int n = 0;
            for (E e : c) {
                if (e == null)
                    throw new NullPointerException();
                if (n == capacity)
                    throw new IllegalStateException("Queue full"
);
                //参考下面的enqueue分析
                enqueue(new Node<E>(e));
                ++n;
            }
            count.set(n);
        } finally {
            //释放锁
            putLock.unlock();
        }
    }

    /**
     * 我去，这代码其实可读性不怎么样啊。
     * 其实下面的代码等价于如下内容:
     * last.next=node;
     * last = node;
     * 其实也没有什么花样:
       就是让新入队列的元素成为原来的last的next，让进入的元素称为last
     *
     */
    private void enqueue(Node<E> node) {
        // assert putLock.isHeldByCurrentThread();
        // assert last.next == null;
```

```
        last = last.next = node;
    }
```

在分析完LinkedBlockingQueue的核心组成之后,下面让我们再看下核心的几个操作方法,首先分析一下元素入队列的过程:

```
    public void put(E e) throws InterruptedException {
        if (e == null) throw new NullPointerException();
        // Note: convention in all put/take/etc is to preset loc
al var

        /*注意上面这句话,约定所有的put/take操作都会预先设置本地变量,
        可以看到下面有一个将putLock赋值给了一个局部变量的操作
        */
        int c = -1;
        Node<E> node = new Node(e);
        /*
         在这里首先获取到putLock,以及当前队列的元素数量
         即上面所描述的预设置本地变量操作
        */
        final ReentrantLock putLock = this.putLock;
        final AtomicInteger count = this.count;
        /*
            执行可中断的锁获取操作,即意味着如果线程由于获取
            锁而处于Blocked状态时，线程是可以被中断而不再继
            续等待，这也是一种避免死锁的一种方式，不会因为
            发现到死锁之后而由于无法中断线程最终只能重启应用。
        */
        putLock.lockInterruptibly();
        try {
            /*
            当队列的容量到底最大容量时,此时线程将处于等待状
            态，直到队列有空闲的位置才继续执行。使用while判
            断依旧是为了放置线程被"伪唤醒”而出现的情况,即当
            线程被唤醒时而队列的大小依旧等于capacity时，线
            程应该继续等待。
            */
            while (count.get() == capacity) {
                notFull.await();
```

```
            }
            //让元素进行队列的末尾,enqueue代码在上面分析过了
            enqueue(node);
            //首先获取原先队列中的元素个数,然后再对队列中的元素个数+1.
            c = count.getAndIncrement();
            /*注:c+1得到的结果是新元素入队列之后队列元素的总和。
            当前队列中的总元素个数小于最大容量时,此时唤醒其他执行入队列的
线程
            让它们可以放入元素,如果新加入元素之后,队列的大小等于capacity
，
            那么就意味着此时队列已经满了,也就没有必须要唤醒其他正在等待入
队列的线程,因为唤醒它们之后，它们也还是继续等待。
            */
            if (c + 1 < capacity)
                notFull.signal();
        } finally {
            //完成对锁的释放
            putLock.unlock();
        }
        /*当c=0时，即意味着之前的队列是空队列,出队列的线程都处于等待状态，
        现在新添加了一个新的元素,即队列不再为空,因此它会唤醒正在等待获取元
素的线程。
        */
        if (c == 0)
            signalNotEmpty();
    }

    /*
    唤醒正在等待获取元素的线程,告诉它们现在队列中有元素了
    */
    private void signalNotEmpty() {
        final ReentrantLock takeLock = this.takeLock;
        takeLock.lock();
        try {
            //通过notEmpty唤醒获取元素的线程
            notEmpty.signal();
        } finally {
            takeLock.unlock();
        }
    }
```

看完put方法，下面再看看下offer是如何处理的方法：

```
    /**
    在BlockingQueue接口中除了定义put方法外(当队列元素满了之后就会阻塞，
    直到队列有新的空间可以方法线程才会继续执行)，还定义一个offer方法，
    该方法会返回一个boolean值，当入队列成功返回true,入队列失败返回false。
    该方法与put方法基本操作基本一致，只是有细微的差异。
    */
     public boolean offer(E e) {
        if (e == null) throw new NullPointerException();
        final AtomicInteger count = this.count;
        /*
            当队列已经满了，它不会继续等待,而是直接返回。
            因此该方法是非阻塞的。
        */
        if (count.get() == capacity)
            return false;
        int c = -1;
        Node<E> node = new Node(e);
        final ReentrantLock putLock = this.putLock;
        putLock.lock();
        try {
            /*
            当获取到锁时，需要进行二次的检查,因为可能当队列的大小为capaci
ty-1时，
            两个线程同时去抢占锁，而只有一个线程抢占成功，那么此时
            当线程将元素入队列后，释放锁，后面的线程抢占锁之后，此时队列
            大小已经达到capacity，所以将它无法让元素入队列。
            下面的其余操作和put都一样，此处不再详述
            */
            if (count.get() < capacity) {
                enqueue(node);
                c = count.getAndIncrement();
                if (c + 1 < capacity)
                    notFull.signal();
            }
        } finally {
            putLock.unlock();
        }
```

```
        if (c == 0)
            signalNotEmpty();
        return c >= 0;
    }
```

BlockingQueue还定义了一个限时等待插入操作，即在等待一定的时间内，如果队列有空间可以插入，那么就将元素入队列，然后返回true,如果在过完指定的时间后依旧没有空间可以插入，那么就返回false，下面是限时等待操作的分析:

```
        /**
         通过timeout和TimeUnit来指定等待的时长
         timeout为时间的长度,TimeUnit为时间的单位
        */
        public boolean offer(E e, long timeout, TimeUnit unit)
        throws InterruptedException {

        if (e == null) throw new NullPointerException();
        //将指定的时间长度转换为毫秒来进行处理
        long nanos = unit.toNanos(timeout);
        int c = -1;
        final ReentrantLock putLock = this.putLock;
        final AtomicInteger count = this.count;
        putLock.lockInterruptibly();
        try {
            while (count.get() == capacity) {
                //如果等待的剩余时间小于等于0，那么直接返回
                if (nanos <= 0)
                    return false;
            /*
              通过condition来完成等待，此时当前线程会完成锁的，并且处于
等待状态
              直到被其他线程唤醒该线程、或者当前线程被中断、
              等待的时间截至才会返回，该返回值为从方法调用到返回所经历的时
长。
              注意：上面的代码是condition的awitNanos()方法的通用写法，
              可以参看Condition.awaitNaos的API文档。
              下面的其余操作和put都一样，此处不再详述
            */
                nanos = notFull.awaitNanos(nanos);
```

```
            }
            enqueue(new Node<E>(e));
            c = count.getAndIncrement();
            if (c + 1 < capacity)
                notFull.signal();
        } finally {
            putLock.unlock();
        }
        if (c == 0)
            signalNotEmpty();
        return true;
    }
```

通过上面的分析，我们应该比较清楚地知道了LinkedBlockingQueue的入队列的操作，其主要是通过获取到putLock锁来完成，当队列的数量达到最大值，此时会导致线程处于阻塞状态或者返回false(根据具体的方法来看)；如果队列还有剩余的空间，那么此时会新创建出一个Node对象，将其设置到队列的尾部，作为LinkedBlockingQueue的last元素。

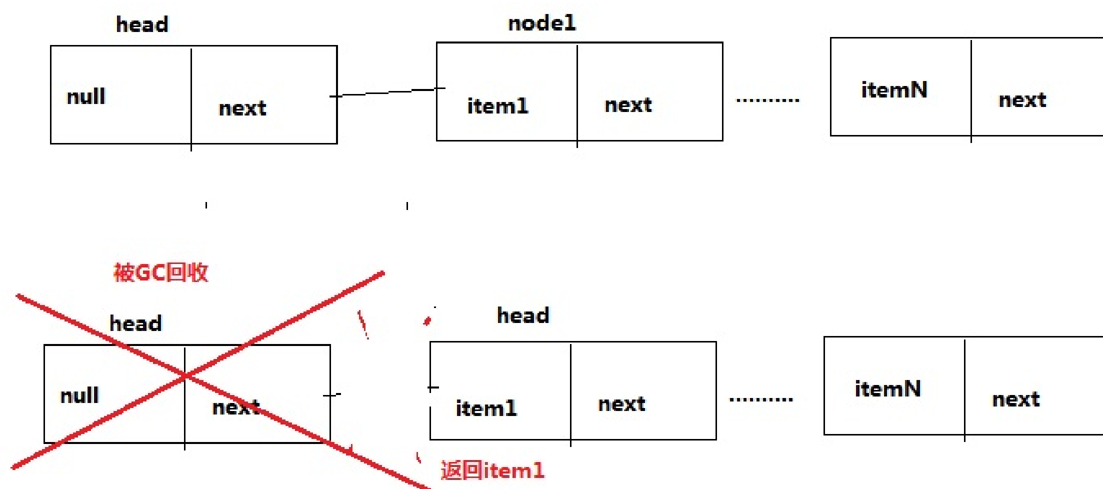
在分析完入队列的过程之后，我们接下来看看LinkedBlockingQueue出队列的过程；由于BlockingQueue的方法都具有对称性，此处就只分析take方法的实现，其余方法的实现都如出一辙：

```
public E take() throws InterruptedException {
        E x;
        int c = -1;
        final AtomicInteger count = this.count;
        final ReentrantLock takeLock = this.takeLock;
        //通过takeLock获取锁，并且支持线程中断
        takeLock.lockInterruptibly();
        try {
            //当队列为空时，则让当前线程处于等待
            while (count.get() == 0) {
                notEmpty.await();
            }
            //完成元素的出队列
            x = dequeue();
            /*
               队列元素个数完成原子化操作-1,可以看到count元素会
```

```
              在插入元素的线程和获取元素的线程进行并发修改操作。
            */
            c = count.getAndDecrement();
            /*
              当一个元素出队列之后，队列的大小依旧大于1时
              当前线程会唤醒其他执行元素出队列的线程,让它们也
              可以执行元素的获取
            */
            if (c > 1)
                notEmpty.signal();
        } finally {
            //完成锁的释放
            takeLock.unlock();
        }
        /*
            当c==capaitcy时，即在获取当前元素之前，
            队列已经满了，而此时获取元素之后，队列就会
            空出一个位置，故当前线程会唤醒执行插入操作的线
            程通知其他中的一个可以进行插入操作。
        */
        if (c == capacity)
            signalNotFull();
        return x;
    }


    /**
     * 让头部元素出队列的过程
     * 其最终的目的是让原来的head被GC回收，让其的next成为head
     * 并且新的head的item为null.
     * 因为LinkedBlockingQueue的头部具有一致性:即元素为null。
     */
    private E dequeue() {
        Node<E> h = head;
        Node<E> first = h.next;
        h.next = h; // help GC
        head = first;
        E x = first.item;
        first.item = null;
        return x;
```

```
    }
```



对于LinkedBlockingQueue的源码分析就到这里，下面让我们将LinkedBlockingQueue与ArrayBlockingQueue进行一个比较。

LinkedBlockingQueue与ArrayBlockingQueue的比较

ArrayBlockingQueue由于其底层基于数组，并且在创建时指定存储的大小，在完成后就会立即在内存分配固定大小容量的数组元素，因此其存储通常有限，故其是一个“有界“的阻塞队列；

而LinkedBlockingQueue可以由用户指定最大存储容量，也可以无需指定，如果不指定则最大存储容量将是Integer.MAX_VALUE，即可以看作是一个“无界”的阻塞队列，由于其节点的创建都是动态创建，并且在节点出队列后可以被GC所回收，因此其具有灵活的伸缩性。但是由于ArrayBlockingQueue的有界性，因此其能够更好的对于性能进行预测，而LinkedBlockingQueue由于没有限制大小，当任务非常多的时候，不停地向队列中存储，就有可能导致内存溢出的情况发生。

其次，ArrayBlockingQueue中在入队列和出队列操作过程中，使用的是同一个lock，所以即使在多核CPU的情况下，其读取和操作的都无法做到并行，而LinkedBlockingQueue的读取和插入操作所使用的锁是两个不同的lock，它们之间的操作互相不受干扰，因此两种操作可以并行完成，故LinkedBlockingQueue的吞吐量要高于ArrayBlockingQueue。

选择LinkedBlockingQueue的理由

```
    /**
        下面的代码是Executors创建固定大小线程池的代码，其使用了
        LinkedBlockingQueue来作为任务队列。
    */
    public static ExecutorService newFixedThreadPool(int nThread
s) {
        return new ThreadPoolExecutor(nThreads, nThreads,
                                      0L, TimeUnit.MILLISECONDS,
                                      new LinkedBlockingQueue<Ru
nnable>());
    }
```

JDK中选用LinkedBlockingQueue作为阻塞队列的原因就在于其无界性。因为线程大小固定的线程池，其线程的数量是不具备伸缩性的，当任务非常繁忙的时候，就势必会导致所有的线程都处于工作状态，如果使用一个有界的阻塞队列来进行处理，那么就非常有可能很快导致队列满的情况发生，从而导致任务无法提交而抛出RejectedExecutionException，而使用无界队列由于其良好的存储容量的伸缩性，可以很好的去缓冲任务繁忙情况下场景，即使任务非常多，也可以进行动态扩容，当任务被处理完成之后，队列中的节点也会被随之被GC回收，非常灵活。

前言

HashMap是我们平时开发过程中用的比较多的集合，但它是非线程安全的，在涉及到多线程并发的情况，进行put操作有可能会引起死循环，导致CPU利用率接近100%。

```
final HashMap<String, String> map = new HashMap<String, String>(2
);
for (int i = 0; i < 10000; i++) {
    new Thread(new Runnable() {
        @Override
        public void run() {
            map.put(UUID.randomUUID().toString(), "");
        }
    }).start();
}
```

解决方案有Hashtable和Collections.synchronizedMap(hashMap)，不过这两个方案基本上是对读写进行加锁操作，一个线程在读写元素，其余线程必须等待，性能可想而知。

所以，Doug Lea给我们带来了并发安全的ConcurrentHashMap，它的实现是依赖于 Java 内存模型，所以我们在了解 ConcurrentHashMap 的之前必须了解一些底层的知识：

1. java内存模型
2. java中的CAS
3. AbstractQueuedSynchronizer
4. ReentrantLock

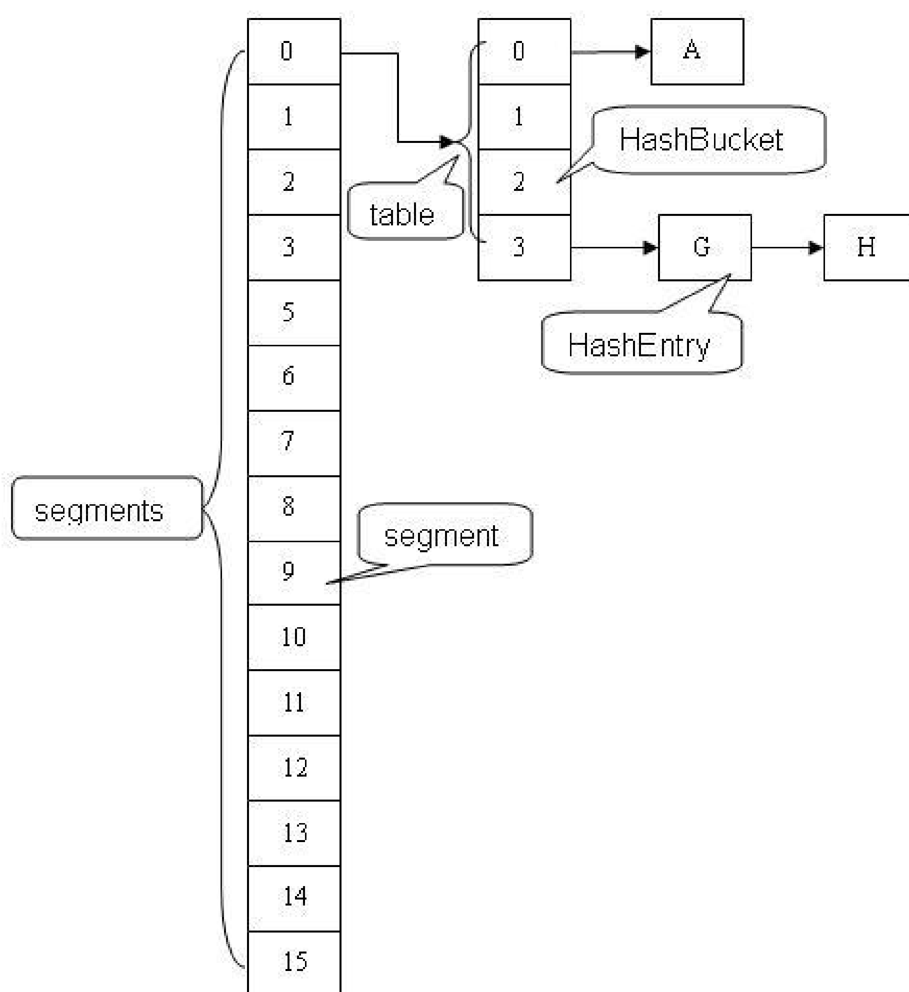
本文源码是JDK8的版本，与之前的版本有较大差异。

JDK1.6分析

ConcurrentHashMap采用 分段锁的机制，实现并发的更新操作，底层采用数组+链表+红黑树的存储结构。 其包含两个核心静态内部类 Segment和HashEntry。

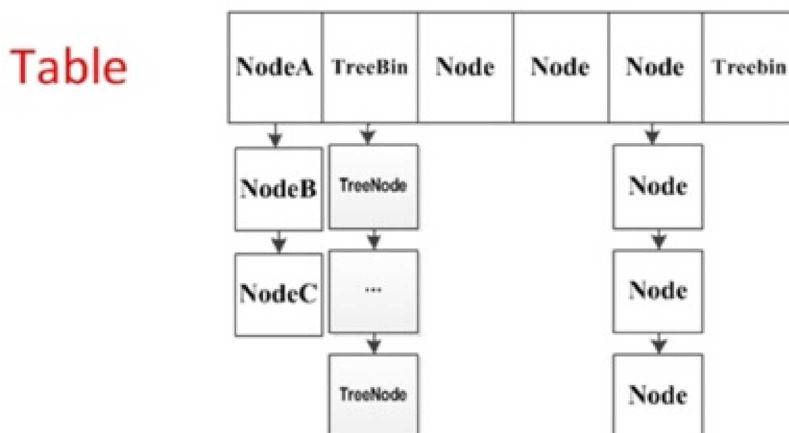
1. Segment继承ReentrantLock用来充当锁的角色，每个 Segment 对象守护每个散列映射表的若干个桶。
2. HashEntry 用来封装映射表的键 / 值对；
3. 每个桶是由若干个 HashEntry 对象链接起来的链表。

一个 ConcurrentHashMap 实例中包含由若干个 Segment 对象组成的数组，下面我们通过一个图来演示一下 ConcurrentHashMap 的结构：



JDK1.8分析

1.8的实现已经抛弃了Segment分段锁机制，利用CAS+Synchronized来保证并发更新的安全，底层依然采用数组+链表+红黑树的存储结构。



重要概念

在开始之前，有些重要的概念需要介绍一下：

1. **table**：默认为null，初始化发生在第一次插入操作，默认大小为16的数组，用来存储Node节点数据，扩容时大小总是2的幂次方。
2. **nextTable**：默认为null，扩容时新生成的数组，其大小为原数组的两倍。
3. sizeCtl

   ：默认为0，用来控制table的初始化和扩容操作，具体应用在后续会体现出来。

   - **-1** 代表table正在初始化
   - **-N** 表示有N-1个线程正在进行扩容操作
   - 其余情况： 1、如果table未初始化，表示table需要初始化的大小。 2、如果table初始化完成，表示table的容量，默认是table大小的0.75倍，居然用这个公式算0.75（n - (n >>> 2)）。
4. Node

   ：保存key，value及key的hash值的数据结构。

```
class Node<K,V> implements Map.Entry<K,V> {
    final int hash;
    final K key;
    volatile V val;
    volatile Node<K,V> next;
    ... 省略部分代码
}
```

其中value和next都用volatile修饰，保证并发的可见性。

5. ForwardingNode

：一个特殊的Node节点，hash值为-1，其中存储nextTable的引用。

```
final class ForwardingNode<K,V> extends Node<K,V> {
    final Node<K,V>[] nextTable;
    ForwardingNode(Node<K,V>[] tab) {
        super(MOVED, null, null, null);
        this.nextTable = tab;
    }
}
```

只有table发生扩容的时候，ForwardingNode才会发挥作用，作为一个占位符放在table中表示当前节点为null或则已经被移动。

实例初始化

实例化ConcurrentHashMap时带参数时，会根据参数调整table的大小，假设参数为100，最终会调整成256，确保table的大小总是2的幂次方，算法如下：

```
ConcurrentHashMap<String, String> hashMap = new ConcurrentHashMa
p<>(100);
private static final int tableSizeFor(int c) {
    int n = c - 1;
    n |= n >>> 1;
    n |= n >>> 2;
    n |= n >>> 4;
    n |= n >>> 8;
    n |= n >>> 16;
    return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ? MAXIMUM_CAPAC
ITY : n + 1;
}
```

注意，ConcurrentHashMap在构造函数中只会初始化sizeCtl值，并不会直接初始化table，而是延缓到第一次put操作。

table初始化

前面已经提到过，table初始化操作会延缓到第一次put行为。但是put是可以并发执行的，Doug Lea是如何实现table只初始化一次的？让我们来看看源码的实现。

```
private final Node<K,V>[] initTable() {
    Node<K,V>[] tab; int sc;
    while ((tab = table) == null || tab.length == 0) {
//如果一个线程发现sizeCtl<0，意味着另外的线程执行CAS操作成功，当前线程只需要让出cpu时间片
        if ((sc = sizeCtl) < 0)
            Thread.yield(); // lost initialization race; just spin
        else if (U.compareAndSwapInt(this, SIZECTL, sc, -1)) {
            try {
                if ((tab = table) == null || tab.length == 0) {
                    int n = (sc > 0) ? sc : DEFAULT_CAPACITY;
                    @SuppressWarnings("unchecked")
                    Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];
                    table = tab = nt;
                    sc = n - (n >>> 2);
                }
            } finally {
                sizeCtl = sc;
            }
            break;
        }
    }
    return tab;
}
```

sizeCtl默认为0，如果ConcurrentHashMap实例化时有传参数，sizeCtl会是一个2的幂次方的值。所以执行第一次put操作的线程会执行Unsafe.compareAndSwapInt方法修改sizeCtl为-1，有且只有一个线程能够修改成功，其它线程通过Thread.yield()让出CPU时间片等待table初始化完成。

put操作

假设table已经初始化完成，put操作采用CAS+synchronized实现并发插入或更新操作，具体实现如下。

```
final V putVal(K key, V value, boolean onlyIfAbsent) {
    if (key == null || value == null) throw new NullPointerExcep
tion();
    int hash = spread(key.hashCode());
    int binCount = 0;
    for (Node<K,V>[] tab = table;;) {
        Node<K,V> f; int n, i, fh;
        if (tab == null || (n = tab.length) == 0)
            tab = initTable();
        else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {
            if (casTabAt(tab, i, null, new Node<K,V>(hash, key,
value, null)))
                break;                   // no lock when adding
to empty bin
        }
        else if ((fh = f.hash) == MOVED)
            tab = helpTransfer(tab, f);
        ...省略部分代码
    }
    addCount(1L, binCount);
    return null;
}
```

1. hash算法

   ```
   static final int spread(int h) {return (h ^ (h >>> 16)) & HA
   SH_BITS;}
   ```

2. table中定位索引位置，n是table的大小

   ```
   int index = (n - 1) & hash
   ```

3. 获取table中对应索引的元素f。 Doug Lea采用Unsafe.getObjectVolatile来获取，也许有人质疑，直接table[index]不可以么，为什么要这么复杂？ 在java内存模型中，我们已经知道每个线程都有一个工作内存，里面存储着table的副

本，虽然table是volatile修饰的，但不能保证线程每次都拿到table中的最新元素，Unsafe.getObjectVolatile可以直接获取指定内存的数据，保证了每次拿到数据都是最新的。

4. 如果f为null，说明table中这个位置第一次插入元素，利用Unsafe.compareAndSwapObject方法插入Node节点。
    - 如果CAS成功，说明Node节点已经插入，随后addCount(1L, binCount)方法会检查当前容量是否需要进行扩容。
    - 如果CAS失败，说明有其它线程提前插入了节点，自旋重新尝试在这个位置插入节点。
5. 如果f的hash值为-1，说明当前f是ForwardingNode节点，意味有其它线程正在扩容，则一起进行扩容操作。
6. 其余情况把新的Node节点按链表或红黑树的方式插入到合适的位置，这个过程采用同步内置锁实现并发，代码如下:

```
synchronized (f) {
 if (tabAt(tab, i) == f) {
     if (fh >= 0) {
         binCount = 1;
         for (Node<K,V> e = f;; ++binCount) {
             K ek;
             if (e.hash == hash &&
                 ((ek = e.key) == key ||
                  (ek != null && key.equals(ek)))) {
                 oldVal = e.val;
                 if (!onlyIfAbsent)
                     e.val = value;
                 break;
             }
             Node<K,V> pred = e;
             if ((e = e.next) == null) {
                 pred.next = new Node<K,V>(hash, key,
                                           value, null);
                 break;
             }
         }
     }
     else if (f instanceof TreeBin) {
         Node<K,V> p;
         binCount = 2;
         if ((p = ((TreeBin<K,V>)f).putTreeVal(hash, key,
                                        value)) != null) {
             oldVal = p.val;
             if (!onlyIfAbsent)
                 p.val = value;
         }
     }
 }
}
```

在节点f上进行同步，节点插入之前，再次利用tabAt(tab, i) == f判断，防止被其它线程修改。

i. 如果f.hash >= 0，说明f是链表结构的头结点，遍历链表，如果找到对应的

node节点，则修改value，否则在链表尾部加入节点。

ii. 如果f是TreeBin类型节点，说明f是红黑树根节点，则在树结构上遍历元素，更新或增加节点。
iii. 如果链表中节点数binCount >= TREEIFY_THRESHOLD(默认是8)，则把链表转化为红黑树结构。

table扩容

当table容量不足的时候，即table的元素数量达到容量阈值sizeCtl，需要对table进行扩容。 整个扩容分为两部分：

1. 构建一个nextTable，大小为table的两倍。
2. 把table的数据复制到nextTable中。

这两个过程在单线程下实现很简单，但是ConcurrentHashMap是支持并发插入的，扩容操作自然也会有并发的出现，这种情况下，第二步可以支持节点的并发复制，这样性能自然提升不少，但实现的复杂度也上升了一个台阶。

先看第一步，构建nextTable，毫无疑问，这个过程只能只有单个线程进行nextTable的初始化，具体实现如下：

```
private final void addCount(long x, int check) {
    ... 省略部分代码
    if (check >= 0) {
        Node<K,V>[] tab, nt; int n, sc;
        while (s >= (long)(sc = sizeCtl) && (tab = table) != null
 &&
               (n = tab.length) < MAXIMUM_CAPACITY) {
            int rs = resizeStamp(n);
            if (sc < 0) {
                if ((sc >>> RESIZE_STAMP_SHIFT) != rs || sc == r
s + 1 ||
                    sc == rs + MAX_RESIZERS || (nt = nextTable)
== null ||
                    transferIndex <= 0)
                    break;
                if (U.compareAndSwapInt(this, SIZECTL, sc, sc + 1
))
                    transfer(tab, nt);
            }
            else if (U.compareAndSwapInt(this, SIZECTL, sc,
                                         (rs << RESIZE_STAMP_SHI
FT) + 2))
                transfer(tab, null);
            s = sumCount();
        }
    }
}
```

通过Unsafe.compareAndSwapInt修改sizeCtl值，保证只有一个线程能够初始化nextTable，扩容后的数组长度为原来的两倍，但是容量是原来的1.5。

节点从table移动到nextTable，大体思想是遍历、复制的过程。

1. 首先根据运算得到需要遍历的次数i，然后利用tabAt方法获得i位置的元素f，初始化一个forwardNode实例fwd。
2. 如果f == null，则在table中的i位置放入fwd，这个过程是采用Unsafe.compareAndSwapObjectf方法实现的，很巧妙的实现了节点的并发移动。

3. 如果f是链表的头节点，就构造一个反序链表，把他们分别放在nextTable的i和i+n的位置上，移动完成，采用Unsafe.putObjectVolatile方法给table原位置赋值fwd。
4. 如果f是TreeBin节点，也做一个反序处理，并判断是否需要untreeify，把处理的结果分别放在nextTable的i和i+n的位置上，移动完成，同样采用Unsafe.putObjectVolatile方法给table原位置赋值fwd。

遍历过所有的节点以后就完成了复制工作，把table指向nextTable，并更新sizeCtl为新数组大小的0.75倍 ，扩容完成。

红黑树构造

注意：如果链表结构中元素超过TREEIFY_THRESHOLD阈值，默认为8个，则把链表转化为红黑树，提高遍历查询效率。

```
if (binCount != 0) {
    if (binCount >= TREEIFY_THRESHOLD)
        treeifyBin(tab, i);
    if (oldVal != null)
        return oldVal;
    break;
}
```

接下来我们看看如何构造树结构，代码如下：

```
private final void treeifyBin(Node<K,V>[] tab, int index) {
    Node<K,V> b; int n, sc;
    if (tab != null) {
        if ((n = tab.length) < MIN_TREEIFY_CAPACITY)
            tryPresize(n << 1);
        else if ((b = tabAt(tab, index)) != null && b.hash >= 0)
 {
            synchronized (b) {
                if (tabAt(tab, index) == b) {
                    TreeNode<K,V> hd = null, tl = null;
                    for (Node<K,V> e = b; e != null; e = e.next)
 {
                        TreeNode<K,V> p =
                            new TreeNode<K,V>(e.hash, e.key, e.v
al,
                                              null, null);
                        if ((p.prev = tl) == null)
                            hd = p;
                        else
                            tl.next = p;
                        tl = p;
                    }
                    setTabAt(tab, index, new TreeBin<K,V>(hd));
                }
            }
        }
    }
}
```

可以看出，生成树节点的代码块是同步的，进入同步代码块之后，再次验证table中index位置元素是否被修改过。 1、根据table中index位置Node链表，重新生成一个hd为头结点的TreeNode链表。 2、根据hd头结点，生成TreeBin树结构，并把树结构的root节点写到table的index位置的内存中，具体实现如下：

```
TreeBin(TreeNode<K,V> b) {
    super(TREEBIN, null, null, null);
    this.first = b;
    TreeNode<K,V> r = null;
```

```
    for (TreeNode<K,V> x = b, next; x != null; x = next) {
        next = (TreeNode<K,V>)x.next;
        x.left = x.right = null;
        if (r == null) {
            x.parent = null;
            x.red = false;
            r = x;
        }
        else {
            K k = x.key;
            int h = x.hash;
            Class<?> kc = null;
            for (TreeNode<K,V> p = r;;) {
                int dir, ph;
                K pk = p.key;
                if ((ph = p.hash) > h)
                    dir = -1;
                else if (ph < h)
                    dir = 1;
                else if ((kc == null &&
                          (kc = comparableClassFor(k)) == null)
||
                         (dir = compareComparables(kc, k, pk)) =
= 0)
                    dir = tieBreakOrder(k, pk);
                    TreeNode<K,V> xp = p;
                if ((p = (dir <= 0) ? p.left : p.right) == null)
 {
                    x.parent = xp;
                    if (dir <= 0)
                        xp.left = x;
                    else
                        xp.right = x;
                    r = balanceInsertion(r, x);
                    break;
                }
            }
        }
    }
    this.root = r;
```

```
        assert checkInvariants(root);
    }
```

主要根据Node节点的hash值大小构建二叉树。这个红黑树的构造过程实在有点复杂，感兴趣的同学可以看看源码。

get操作

get操作和put操作相比，显得简单了许多。

```
public V get(Object key) {
    Node<K,V>[] tab; Node<K,V> e, p; int n, eh; K ek;
    int h = spread(key.hashCode());
    if ((tab = table) != null && (n = tab.length) > 0 &&
        (e = tabAt(tab, (n - 1) & h)) != null) {
        if ((eh = e.hash) == h) {
            if ((ek = e.key) == key || (ek != null && key.equals
(ek)))
                return e.val;
        }
        else if (eh < 0)
            return (p = e.find(h, key)) != null ? p.val : null;
        while ((e = e.next) != null) {
            if (e.hash == h &&
                ((ek = e.key) == key || (ek != null && key.equal
s(ek))))
                return e.val;
        }
    }
    return null;
}
```

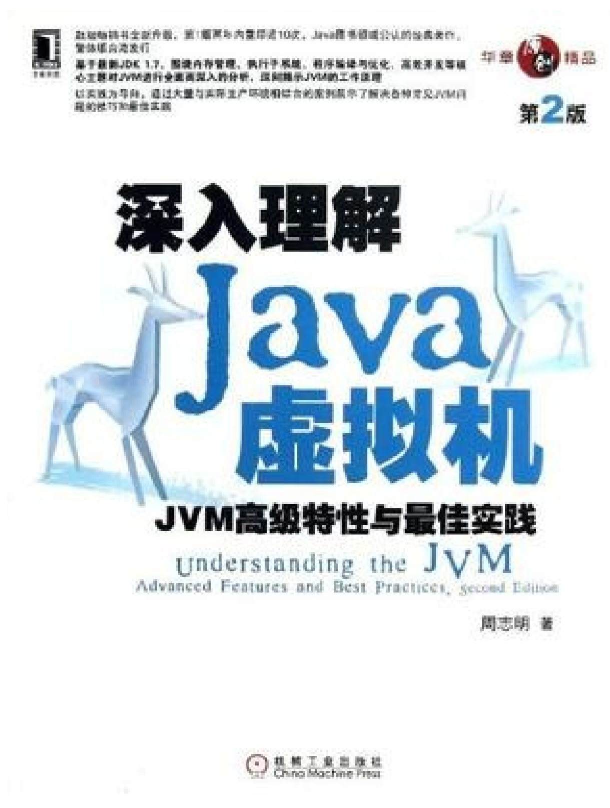
1. 判断table是否为空，如果为空，直接返回null。
2. 计算key的hash值，并获取指定table中指定位置的Node节点，通过遍历链表或则树结构找到对应的节点，返回value值。

总结

ConcurrentHashMap 是一个并发散列映射表的实现，它允许完全并发的读取，并且支持给定数量的并发更新。相比于 HashTable 和同步包装器包装的 HashMap，使用一个全局的锁来同步不同线程间的并发访问，同一时间点，只能有一个线程持有锁，也就是说在同一时间点，只能有一个线程能访问容器，这虽然保证多线程间的安全并发访问，但同时也导致对容器的访问变成串行化的了。

一、前言

本部分内容是关于Java虚拟机的一些面试高频知识点的总结。说到对Java虚拟机的学习，就不得不提下这本书《深入理解Java虚拟机》。



本部分的内容也是基于这本书进行整理的，这本书基本是面试必备。

关于Java虚拟机，重点考察以下三个方面的内容：

- 内存区域/内存模型
- 类加载机制
- 垃圾收集算法/收集器

二、目录

一、对象的创建

1.虚拟机遇到一个new指令时，首先将去检查这个指令的参数是否能在常量池中定位到一个类的符号引用；

2.检查这个符号引用代表的类是否已经被加载，解析和初始化过。如果没有，那必须先执行响应的类加载过程；

3.在类加载检查功通过后，为新生对象分配内存。对象所需的内存大小在类加载完成后便可完全确定。

二、对象的内存布局

分为**3**个区域：对象头，实例数据，对齐填充。

对象头：

包括两部分信息，第一部分：对象自身的运行时数据，如哈希码，GC分代年龄，锁状态标志，线程持有的锁，偏向线程ID，偏向时间戳等，这部分数据的长度在32位和64位的虚拟机中分别为32 bit和64 bit，官方称它为“Mark Word”。

第二部分：类型指针，即对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例。如果对象是一个java数组，那在对象头中还必须有一块用于记录数组长度的数据。

实例数据：

是对象眞正存储的有效信息，也是在程序代码中所定义的各种类型的字段内容。

对齐填充：

对齐填充不是必然存在的。HotSpot VM的自动内存管理系统要求对象起始地址必须是8字节的整数倍，也就是说对象的大小必须是8字节的整数倍。而对象头部分正好是8字节的整数倍。因此，当对象实例数据部分没有对齐时，就需要通过对其补充来补全了。

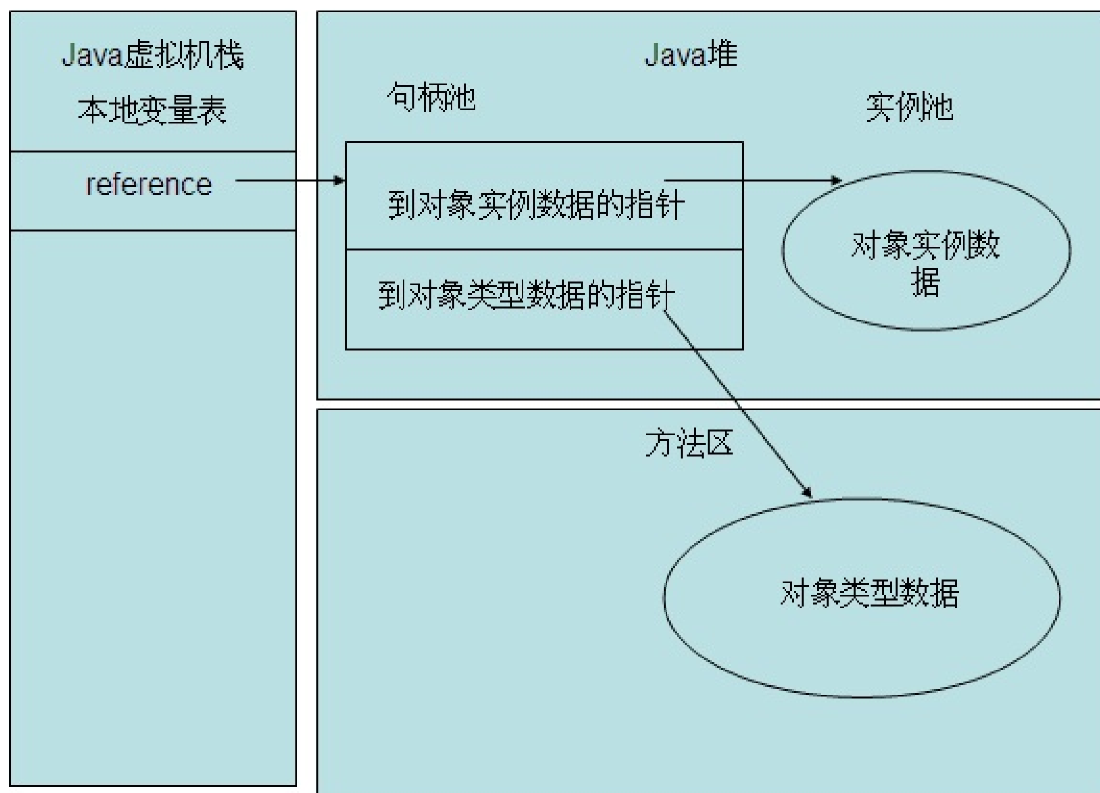
三、对象的访问定位

Java程序需要通过栈上了reference数据来操作堆上的具体对象。

目前主流的访问方式有使用句柄和直接指针两种。

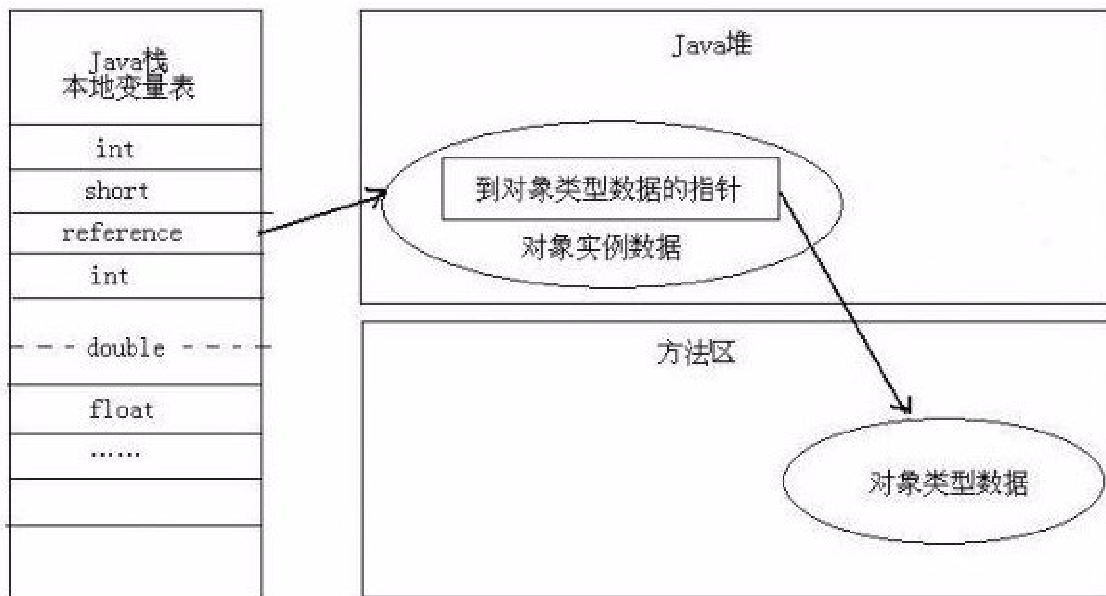
句柄访问：

Java堆中会划分出一块内存来作为句柄池，reference中存储的就是对象的句柄地址，而句柄中包含了对实例数据与类型数据的各自具体的地址信息。



直接指针访问：

reference中存储的直接就是对象地址。


Java栈
本地变量表
int
short
reference
int
double
float
……
Java堆
到对象类型数据的指针
对象实例数据
方法区
对象类型数据

一、Java内存区域

方法区（公有）：用户存储已被虚拟机加载的类信息，常量，静态常量，即时编译器编译后的代码等数据。异常状态 OutOfMemoryError

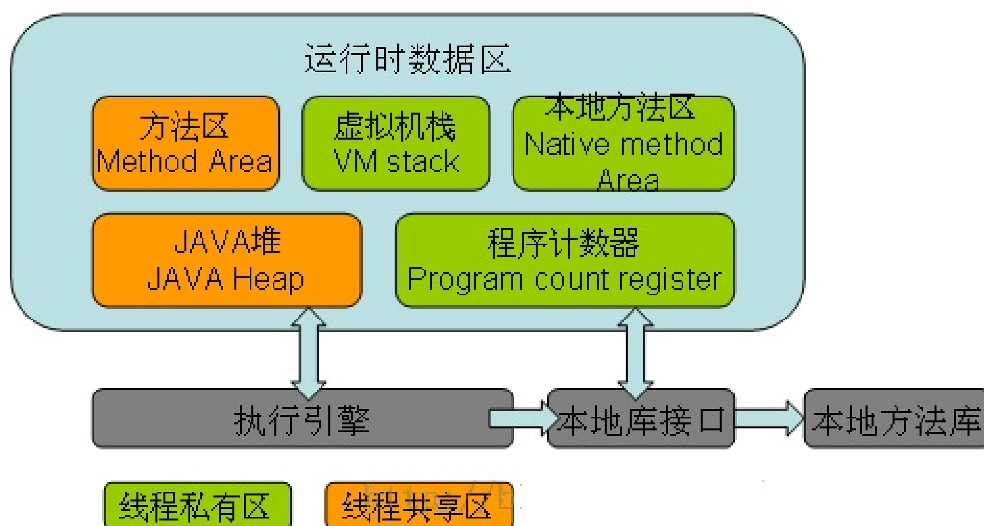
其中包含常量池：用户存放编译器生成的各种字面量和符号引用。

堆（公有）：是JVM所管理的内存中最大的一块。唯一目的就是存放实例对象，几乎所有的对象实例都在这里分配。Java堆是垃圾收集器管理的主要区域，因此很多时候也被称为“GC堆”。异常状态 OutOfMemoryError

虚拟机栈（线程私有）：描述的是java方法执行的内存模型：每个方法在执行时都会创建一个栈帧，用户存储局部变量表，操作数栈，动态连接，方法出口等信息。每一个方法从调用直至完成的过程，就对应着一个栈帧在虚拟机栈中入栈到出栈的过程。 对这个区域定义了两种异常状态 OutOfMemoryError StackOverflowError

本地方法栈（线程私有）：与虚拟机栈所发挥的作用相似。它们之间的区别不过是虚拟机栈为虚拟机执行java方法，而本地方法栈为虚拟机使用到的Native方法服务。

程序计数器（线程私有）：一块较小的内存，当前线程所执行的字节码的行号指示器。字节码解释器工作时，就是通过改变这个计数器的值来选取下一条需要执行的字节码指令。

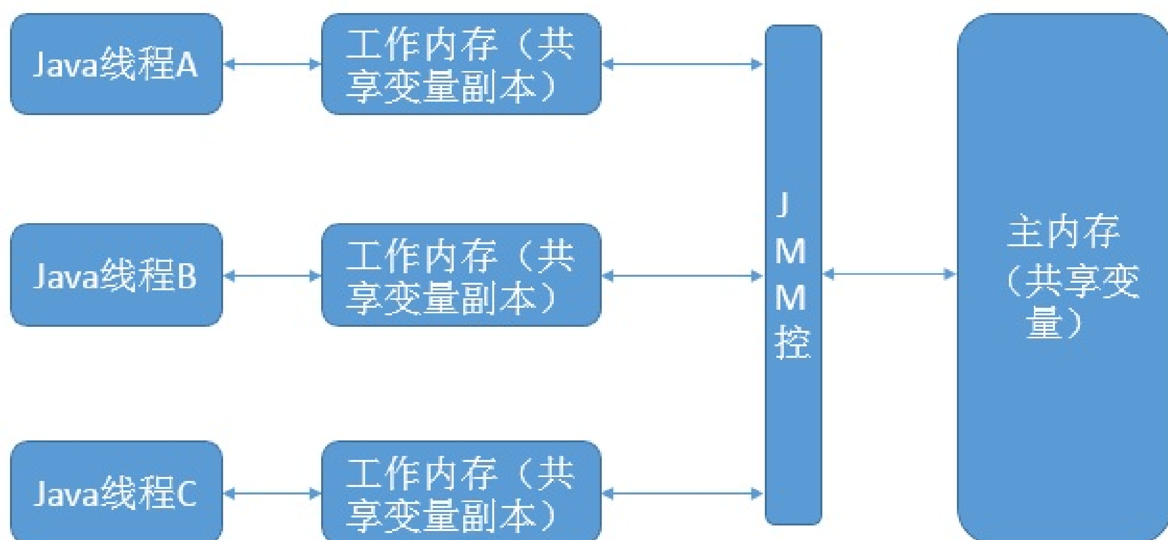


二、Java内存模型

Java内存模型的目的： 屏蔽掉各种硬件和操作系统的内存访问差异，以实现让java程序在各种平台下都能达到一致的内存访问效果。

主要目标： 定义程序中各个变量的访问规则，即在虚拟机中将变量存储到内存和从内存中取出变量这样的底层细节。此处的变量与Java变成中所说的变量是有所区别，它包括了实例字段，静态字段和构成数组对象的元素，但不包括局部变量和方法参数。

Java内存模型规定了所有的变量都存储在主内存中。每条线程中还有自己的工作内存，线程的工作内存中保存了被该线程所使用到的变量（这些变量是从主内存中拷贝而来）。线程对变量的所有操作（读取，赋值）都必须在工作内存中进行。不同线程之间也无法直接访问对方工作内存中的变量，线程间变量值的传递均需要通过主内存来完成。



关于主内存与工作内存之间具体的交互协议，即一个变量如何从主内存拷贝到工作内存、如何从工作内存同步回主内存之类的实现细节，Java内存模型中定义了8种操作来完成，并且每种操作都是原子的、不可再分的。

八种操作：

类型	说明
lock	作用于主内存的变量，把一个变量标识为一条线程独占的状态
unlock	作用于主内存的变量，把一个处于锁定状态的变量释放出来。
read	把一个变量的值从主内存传输到工作内存中，以便随后的load使用。
load	把read操作从主内存中得到的变量值放入到工作内存的变量副本中。
use	把工作内存中一个变量的值传递给执行引擎，每当虚拟机遇到一个需要使用变量的值的字节码指令时将会执行这个操作。
assign	把一个从执行引擎中接收到的值赋值给工作内存中的变量，每当虚拟机遇到一个给变量赋值的字节码指令时执行这个操作。
store	把工作内存中的一个变量的值传递到主内存，以便随后的write使用。
write	把store操作从工作内存中得到的变量值放入到主内存的变量中。

一、类加载机制

1.定义：

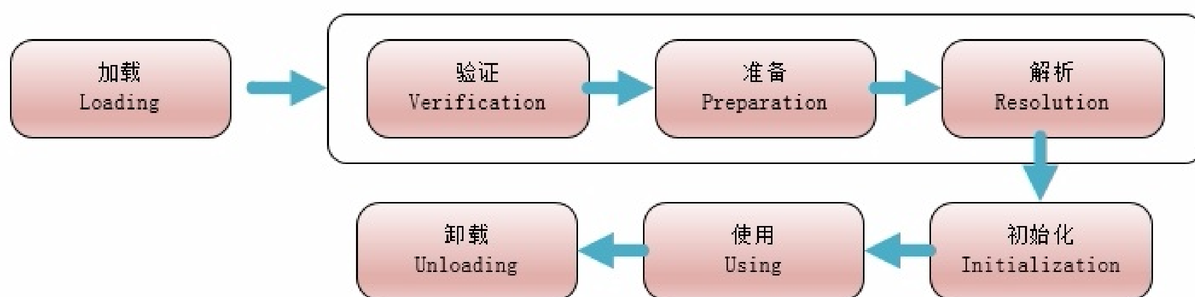
把描述类的数据从Class文件加载到内存，并对数据进行校验、转换解析和初始化，最终形成可以被虚拟机直接使用的Java类型。

在Java语言里，类型的加载、连接和初始化过程都是在程序运行期间完成的，这种策略虽然会令类加载时稍微增加一些性能开销，但是会为Java应用程序提供高度的灵活性，Java里天生可以动态扩展的语言特性就是依赖运行期动态加载和动态连接这个特点来实现的。

2.类的生命周期：

加载，验证，准备，解析，初始化，使用和卸载。其中验证，准备，解析3个部分统称为连接。

这7个阶段发生顺序如下图：



加载，验证，准备，初始化，卸载这5个阶段的顺序是确定的，而解析阶段则不一定：它在某些情况下可以在初始化完成后在开始，这是为了支持Java语言的运行时绑定。

其中加载，验证，准备，解析及初始化是属于类加载机制中的步骤。注意此处的加载不等同于类加载。

3.触发类加载的条件：

①.遇到new,getstatic,putstatic或invokestatic这4条字节码指令时，如果类没有进行过初始化，则需要先触发初始化。生成这4条指令的最常见的Java代码场景是：使用new关键字实例化对象的时候，读取或设置一个类的静态字段的时候（被final修饰，已在编译期把结果放入常量池的静态字段除外），以及调用一个类的静态方法的时候。

②.使用java.lang.reflect包的方法对类进行反射调用的时候。

③.当初始化一个类的时候，发现其父类还没有进行过初始化，则需要先出发父类的初始化。

④.当虚拟机启动时，用户需要指定一个要执行的主类（包含main()方法的那个类），虚拟机会先初始化这个主类。

⑤.当使用JDK1.7的动态语言支持时，如果一个 `java.lang.invoke.MethodHandle` 实例最后的解析结果 `REF_getStatic,REF_putStatic,REF_invokeStatic` 的方法句柄，并且这个方法句柄所对应的类没有进行初始化，则需要先出发初始化。

4.类加载的具体过程：

加载:

①.通过一个类的全限定名来获取定义此类的二进制字节流

②.将这个字节流所代表的静态存储结构转换为方法区内的运行时数据结构

③.在内存中生成一个代表这个类的java.lang.Class对象，作为方法区这个类的各种数据的访问入口。

验证：

是连接阶段的第一步，目的是为了确保Class文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。

包含四个阶段的校验动作

a.文件格式验证

验证字节流是否符合Class文件格式的规范，并且能被当前版本的虚拟机处理。

b.元数据验证

对类的元数据信息进行语义校验，是否不存在不符合Java语言规范的元数据信息

c.字节码验证

最复杂的一个阶段，主要目的是通过数据流和控制流分析，确定程序语义是合法的，符合逻辑的。对类的方法体进行校验分析，保证被校验类的方法在运行时不会做出危害虚拟机安全的事件。

d.符号引用验证

最后一个阶段的校验发生在虚拟机将符号引用转换为直接引用的时候，这个转换动作将在连接的第三个阶段——解析阶段中发生。

符号验证的目的是确保解析动作能正常进行。

准备：

准备阶段是正式为类变量分配内存并设置类变量初始值的阶段。这些变量所使用的内存都将在方法区中分配。只包括类变量。初始值“通常情况”下是数据类型的零值。

“特殊情况”下，如果类字段的字段属性表中存在ConstantValue属性，那么在准备阶段变量的值就会被初始化为ConstantValue属性所指定的值。

解析：

虚拟机将常量池内的符号引用替换为直接引用的过程。

“动态解析”的含义就是必须等到程序实际运行到这条指令的时候，解析动作才能进行。相对的，其余可触发解析的指令都是“静态”的，可以在刚刚完成加载阶段，还没有开始执行代码时就进行解析。

初始化：

类加载过程中的最后一步。

初始化阶段是执行类构造器 `<clinit>()` 方法的过程。

`<clinit>()` 方法是由编译器自动收集类中的所有类变量的赋值动作和静态语句块中的语句合并产生的。

`<clinit>()` 与类的构造函数不同，它不需要显示地调用父类构造器，虚拟机会保证在子类的 `<clinit>()` 方法执行之前，父类的 `<clinit>()` 方法已经执行完毕。

简单地说，初始化就是对类变量进行赋值及执行静态代码块。

二、类加载器

通过上述的了解，我们已经知道了类加载机制的大概流程及各个部分的功能。其中加载部分的功能是将类的class文件读入内存，并为之创建一个java.lang.Class对象。这部分功能就是由类加载器来实现的。

1.类加载器分类：

不同的类加载器负责加载不同的类。主要分为两类。

启动类加载器（**Bootstrap ClassLoader**）：由C++语言实现（针对HotSpot）,负责将存放在\lib目录或-Xbootclasspath参数指定的路径中的类库加载到内存中，即负责加载Java的核心类。

其他类加载器：由Java语言实现，继承自抽象类ClassLoader。如：

扩展类加载器（**Extension ClassLoader**）：负责加载\lib\ext目录或java.ext.dirs系统变量指定的路径中的所有类库，即负责加载Java扩展的核心类之外的类。

应用程序类加载器（**Application ClassLoader**）：负责加载用户类路径（classpath）上的指定类库，我们可以直接使用这个类加载器，通过ClassLoader.getSystemClassLoader()方法直接获取。一般情况，如果我们没有自定义类加载器默认就是用这个加载器。

以上2大类，3小类类加载器基本上负责了所有Java类的加载。下面我们来具体了解上述几个类加载器实现类加载过程时相互配合协作的流程。

2.双亲委派模型

双亲委派模型的工作流程是：如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把请求委托给父加载器去完成，依次向上，因此，所有的类加载请求最终都应该被传递到顶层的启动类加载器中，只有当父加载器在它的搜索范围中没有找到所需的类时，即无法完成该加载，子加载器才会尝试自己去加载该类。



这样的好处是不同层次的类加载器具有不同优先级，比如所有Java对象的超级父类java.lang.Object，位于rt.jar，无论哪个类加载器加载该类，最终都是由启动类加载器进行加载，保证安全。即使用户自己编写一个java.lang.Object类并放入程序中，虽能正常编译，但不会被加载运行，保证不会出现混乱。

3.双亲委派模型的代码实现

ClassLoader中loadClass方法实现了双亲委派模型

```
protected Class<?> loadClass(String name, boolean resolve)
    throws ClassNotFoundException
{
    synchronized (getClassLoadingLock(name)) {
        //检查该类是否已经加载过
        Class c = findLoadedClass(name);
        if (c == null) {
            //如果该类没有加载，则进入该分支
            long t0 = System.nanoTime();
            try {
                if (parent != null) {
                    //当父类的加载器不为空，则通过父类的loadClass来加载
```

```
该类
                        c = parent.loadClass(name, false);
                    } else {
                        //当父类的加载器为空，则调用启动类加载器来加载该类
                        c = findBootstrapClassOrNull(name);
                    }
                } catch (ClassNotFoundException e) {
                    //非空父类的类加载器无法找到相应的类，则抛出异常
                }

                if (c == null) {
                    //当父类加载器无法加载时，则调用findClass方法来加载该类
                    long t1 = System.nanoTime();
                    c = findClass(name); //用户可通过覆写该方法，来自定义
类加载器

                    //用于统计类加载器相关的信息
                    sun.misc.PerfCounter.getParentDelegationTime().a
ddTime(t1 - t0);
                    sun.misc.PerfCounter.getFindClassTime().addElaps
edTimeFrom(t1);
                    sun.misc.PerfCounter.getFindClasses().increment(
);
                }
            }
            if (resolve) {
                //对类进行link操作
                resolveClass(c);
            }
            return c;
        }
}
```

整个流程大致如下：

a.首先，检查一下指定名称的类是否已经加载过，如果加载过了，就不需要再加载，直接返回。

b.如果此类没有加载过，那么，再判断一下是否有父加载器；如果有父加载器，则由父加载器加载（即调用parent.loadClass(name, false);）.或者是调用bootstrap类加载器来加载。

c.如果父加载器及bootstrap类加载器都没有找到指定的类，那么调用当前类加载器的findClass方法来完成类加载。

关于自定义类加载器，本篇文章就不介绍了，主要是重写findClass方法，有兴趣的可以参考这篇文章。

一、垃圾收集算法

1.标记-清除算法

最基础的收集算法是“标记-清除”（Mark-Sweep）算法，如同它的名字一样，算法分为“标记”和“清除”两个阶段。

①首先标记出所有需要回收的对象

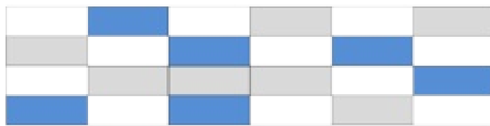
②在标记完成后统一回收所有被标记的对象。

不足：

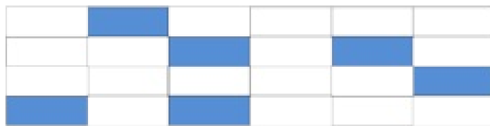
效率问题：标记和清除两个过程的效率都不高

空间问题：标记清除之后产生大量不连续的内存碎片，空间碎片太多可能会导致以后程序运行过程中需要分配较大对象时，无法找到足够的连续内存而不得不提前触发另一次垃圾收集动作。

内存整理前



内存整理后



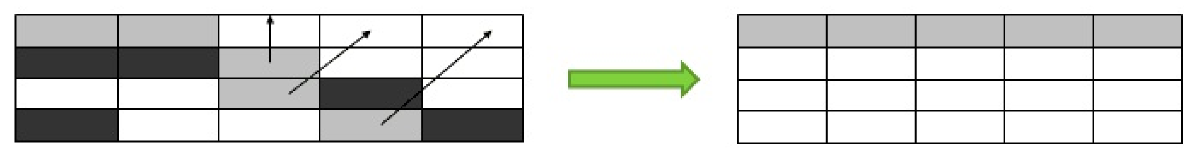


2.复制算法

目的： 为了解决效率问题。

将可用内存按容量大小划分为大小相等的两块，每次只使用其中的一块。当一块内存使用完了，就将还存活着的对象复制到另一块上面，然后再把已使用过的内存空间一次清理掉。这样使得每次都是对整个半区进行内存回收，内存分配时也就不用考虑内存碎片等复杂情况。

缺点： 将内存缩小为了原来的一半。

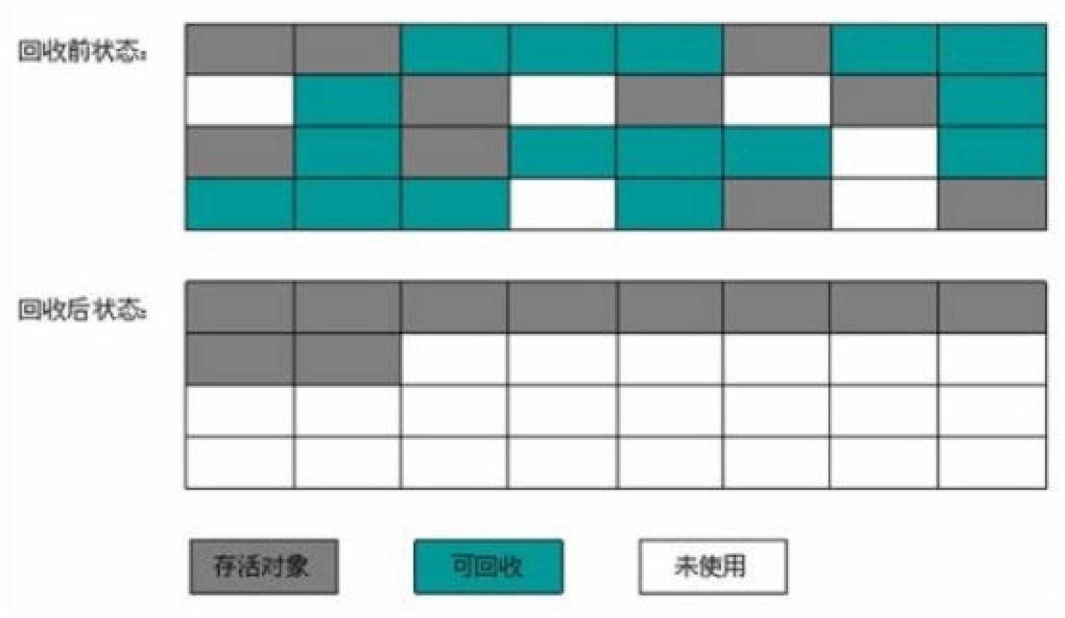


现代的商业虚拟机都采用这种收集算法来回收新生代，IBM公司的专门研究表明，新生代中对象98%对象是“朝生夕死”的，所以不需要按照1：1的比例来划分内存空间，而是将内存分为较大的**Eden**空间和两块较小的**Survivor**空间，每次使用**Eden**和其中一块**Survivor**。**HotSpot**虚拟机中默认**Eden**和**Survivor**的大小比例是**8**：**1**。

3.标记-整理算法

复制收集算法在对象存活率较高时，就要进行较多的复制操作，效率就会变低。 根据老年代的特点，提出了“标记-整理”算法。

标记过程仍然与”标记-清除“算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理掉边界以外的内存。



4.分代收集算法

一般是把Java堆分为新生代和老年代，这样就可以根据各个年代的特点采用最适当的收集算法。

在新生代中，每次垃圾收集时都发现有大批对象死去，只有少量存活，那就选用复制算法。

在老年代中，因为对象存活率高、没有额外空间对它进行分配担保，就必须采用“标记-清除”或“标记-整理”算法来进行回收。

二、垃圾回收机制的一些知识

1.JVM中的年代

JVM中分为年轻代（**Young generation**）和老年代**(Tenured generation)**。

HotSpot JVM把年轻代分为了三部分：1个Eden区和2个Survivor区（分别叫from和to）。默认比例为8：1，为啥默认会是这个比例，接下来我们会聊到。

一般情况下，新创建的对象都会被分配到**Eden**区(一些大对象特殊处理)，这些对象经过第一次Minor GC后，如果仍然存活，将会被移到Survivor区。对象在Survivor区中每熬过一次Minor GC，年龄就会增加1岁，当它的年龄增加到一定程度时，就会被移动到年老代中。 因为年轻代中的对象基本都是朝生夕死的(80%以上)，所以在年轻代的垃圾回收算法使用的是复制算法，复制算法的基本思想就是将内存分为两块，每次只用其中一块，当这一块内存用完，就将还活着的对象复制到另外一块上面。复制算法不会产生内存碎片。

在GC开始的时候，对象只会存在于Eden区和名为“From”的Survivor区，Survivor区“To”是空的。紧接着进行GC，Eden区中所有存活的对象都会被复制到“To”，而在“From”区中，仍存活的对象会根据他们的年龄值来决定去向。年龄达到一定值(年龄阈值，可以通过-XX:MaxTenuringThreshold来设置)的对象会被移动到年老代中，没有达到阈值的对象会被复制到“To”区域。经过这次GC后，Eden区和From区已经被清空。这个时候，“From”和“To”会交换他们的角色，也就是新的“To”就是上次GC前的“From”，新的“From”就是上次GC前的“To”。不管怎样，都会保证名为To的Survivor区域是空的。Minor GC会一直重复这样的过程，直到“To”区被填满，“To”区被填满之后，会将所有对象移动到年老代中。

2.Minor GC和Full GC的区别

Minor GC:指发生在新生代的垃圾收集动作，该动作非常频繁。

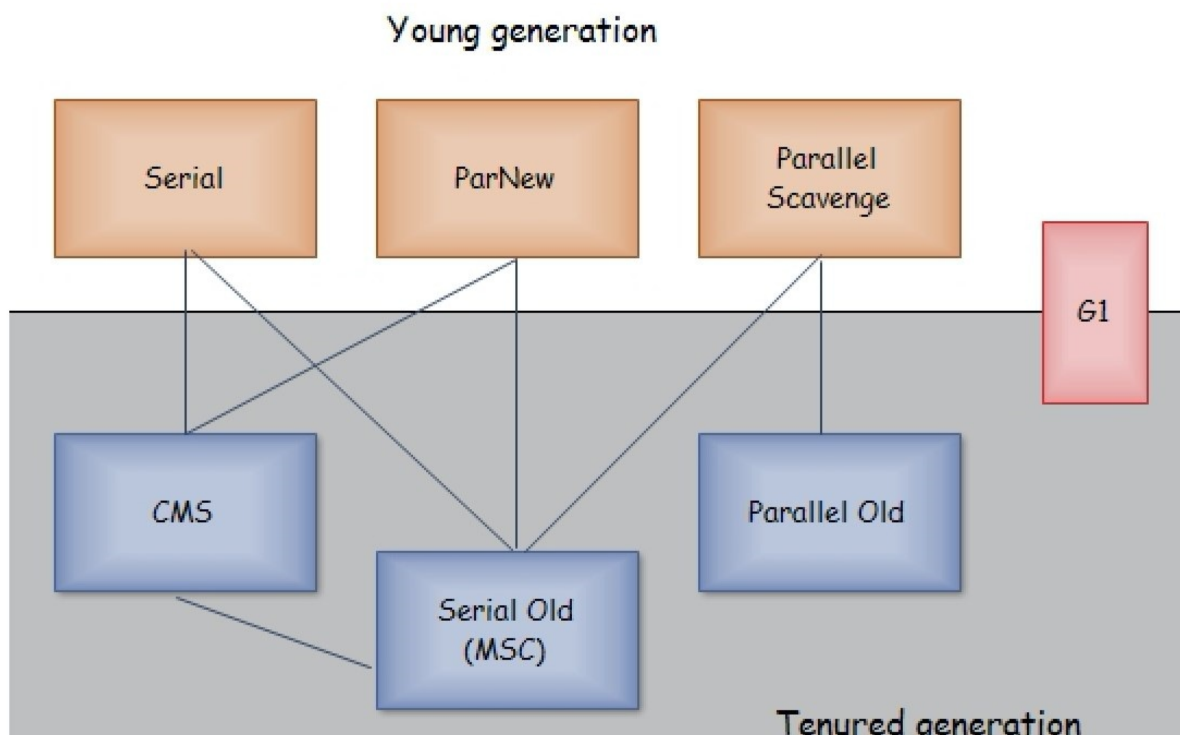
Full GC/Major GC:指发生在老年代的垃圾收集动作，出现了Major GC，经常会伴随至少一次的Minor GC。Major GC的速度一般会比Minor GC慢10倍以上。

3. 空间分配担保

在发生Minor GC之前，虚拟机会先检查老年代最大可用的连续空间是否大于新生代所有对象的总空间，如果这个条件成立，那么Minor GC可以 确保是安全的。如果不成立，则虚拟机会查看**HandlePromotionFailure**设置值是否允许担保失败。如果允许，那会继续检查老年代最大可用的连续空间是否大于历次晋升到老年代对象的平均大小，如果大于，则将尝试进行一次**Minor GC**，尽管这个Minor GC是有风险的。如果小于，或者HandlePromotionFailure设置不允许冒险，那这时也要改为进行一次Full GC。

以上便是在垃圾回收过程中，需要了解的一些必要的知识。下面我们就来介绍具体的垃圾收集器。

三、垃圾收集器



上图展示了7种作用于不同分代的收集器，如果两个收集器之间存在连线，说明它们可以搭配使用。

1.Serial收集器

是最基本、发展历史最悠久的收集器。这是一个单线程收集器。但它的“单线程”的意义并不仅仅说明它只会使用一个CPU或一条收集线程去完成垃圾收集工作，更重要的是它在进行垃圾收集时，必须暂停其他所有的工作线程，直到它收集结束。



Serial / Serial Old 收集器运行示意图

是虚拟机运行在Client模式下的默认新生代收集器。

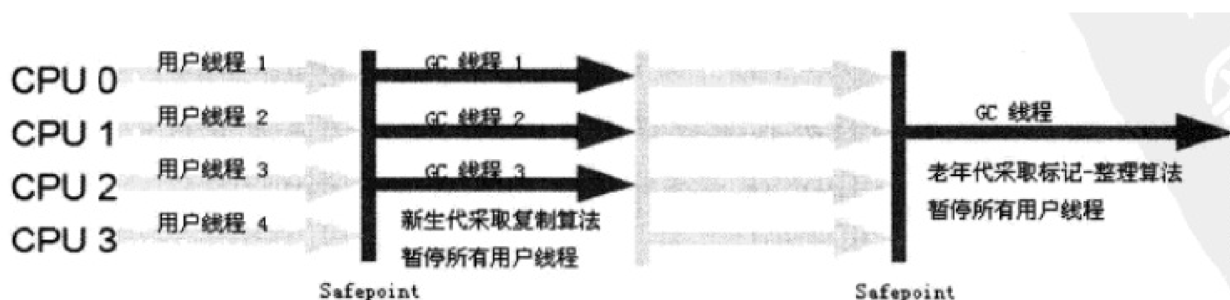
优势：简单而高效（与其他收集器的单线程比），对于限定单个CPU的环境来说，Serial收集器由于没有线程交互的开销，专心做垃圾收集自然可以获得最高的单线程效率。

2.ParNew收集器

ParNew收集器其实就是Serial收集器的多线程版本。

是许多运行在Server模式下的虚拟机中首选的新生代收集器，其中一个与性能无关但很重要的原因是，除了Serial收集器外，目前只有它能与CMS收集器配合工作。

ParNew收集器默认开启的收集线程数与CPU的数量相同。 下图是ParNew/Serial Old收集器运行示意图



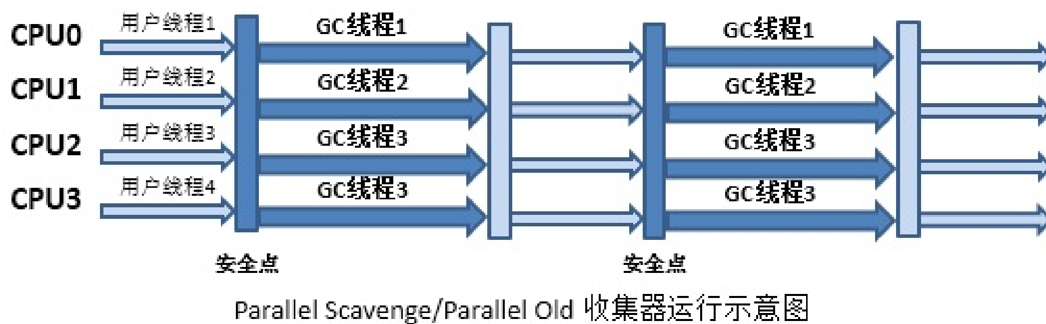
3.Parallel Scavenge收集器

Parallel Scavenge收集器是一个新生代收集器，使用复制算法，又是并行的多线程收集器。

最大的特点是： Parallel Scavenge收集器的目标是达到一个可控制的吞吐量。

所谓吞吐量就是**CPU**用于运行用户代码的时间与**CPU**总消耗时间的比值，即吞吐量=运行用户代码时间/（运行用户代码时间+垃圾收集时间）。

高吞吐量则可以高效率地利用CPU时间，尽快完成程序的运算任务，主要适合在后台运算而不需要太多交互的任务。



Parallel Scavenge/Parallel Old 收集器运行示意图

4.Serial Old收集器

Serial Old是Serial收集器的老年代版本，同样是一个单线程收集器，使用“标记-整理”算法。这个收集器的主要意义也是在于给Client模式下虚拟机使用。

如果在Server模式下，它主要还有两大用途：

1.与Parallel Scavenge收集器搭配使用

2.作为CMS收集器的后备预案，在并发收集发生Conurrent Mode Failure使用。

5.Parallel Old收集器

Parallel Old是Parallel Scavenge收集器的老年代版本，使用多线程和“标记-整理”算法。

在注重吞吐量以及**CPU**资源敏感的场合，都可以优先考虑**Parallel Scavenge+Parallel Old**收集器

6.CMS（Concurrent Mark Sweep）收集器

是**HotSpot**虚拟机中第一款真正意义上的并发收集器，它第一次实现了让垃圾收集线程与用户线程同时工作。

关注点：尽可能地缩短垃圾收集时用户线程的停顿时间。

CMS收集器是基于“标记-清除”算法实现的，整个过程分为**4**个步骤：

①初始标记

②并发标记

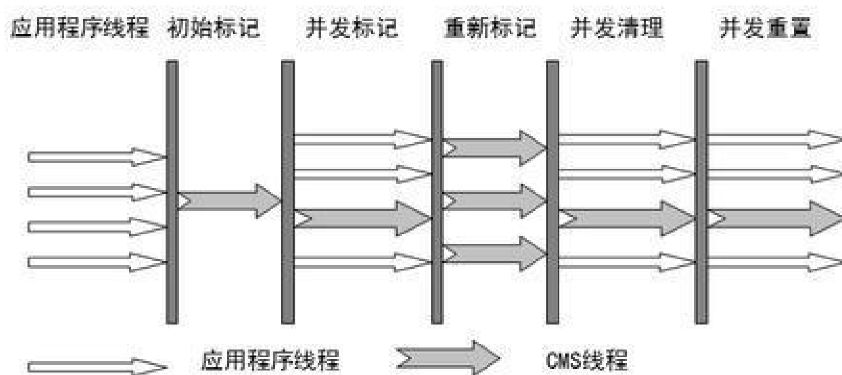
③重新标记

④并发清除

其中，初始标记，重新标记这两个步骤仍然需要“Stop The World”。初始标记仅仅只标记一下GC Roots能直接关联到的对象，速度很快。并发标记阶段就是 进行GC Roots Tracing的过程。

重新标记阶段则是为了修正并发标记期间因用户程序继续运作而导致标记产生变动的那一部分对象的标记几率，这个阶段的停顿时间一般会比初始标记阶段稍长，但远比并发标记时间短。

整个过程耗时最长的阶段是并发标记，并发清除过程，但这两个过程可以和用户线程一起工作。



缺点：

①CMS收集器对CPU资源非常敏感。在并发阶段，它虽然不会导致用户线程停顿，但是会因为占用了一部分线程（或者说CPU资源）而导致应用程序变慢，总吞吐量会降低。

②CMS收集器无法处理浮动垃圾，可能出现“Conurrent Mode Failure”失败而导致另一次Full GC的产生。由于CMS并发清理阶段用户线程还在运行着，伴随程序运行自然就还会产生新的垃圾，这一部分垃圾出现在标记过程之后，CMS无法在档次收集中处理掉它们，只好留待下一次GC时再清理掉。这部分垃圾就称为“浮动垃圾”。因此CMS收集器不能像其他收集器那样等到老年代几乎完全被填满了再进行收集，需要预留一部分空间提供并发收集时程序运作使用。在JDK1.5的默认设置下，CMS收集器当老年代使用了68%的空间后就会被激活。如果预留空间无法满足程序需要，就会出现一次“Concurrent Mode Failure”失败，这时虚拟机将启动后备预案Serial Old。

③CMS是一款基于“标记-清除”算法实现的收集器，所以会有大量空间碎片问题。

7.G1收集器

是当今收集器技术发展的最前沿成果之一。是一款面向服务端应用的垃圾收集器。

特点：

①并行与并发

能充分利用多CPU，多核环境下的硬件优势，缩短Stop-The-World停顿的时间，同时可以通过并发的方式让Java程序继续执行

②分代收集

可以不需要其他收集器的配合管理整个堆，但是仍采用不同的方式去处理分代的对象。

③空间整合

G1从整体上来看，采用基于“标记-整理”算法实现收集器

G1从局部上来看，采用基于“复制”算法实现。

④可预测停顿

使用G1收集器时，Java堆内存布局与其他收集器有很大差别，它将整个Java堆划分成为多个大小相等的独立区域。 G1跟踪各个Region里面的垃圾堆积的价值大小（回收所获得的空间大小以及回收所需时间的经验值），在后台维护一个优先列表，每次根据允许的收集时间，优先回收价值最大的Region。

时间： 2018-05-12 01:44:07

一、引用计数法

给对象添加一个引用计数器，每当有一个地方引用它时，计数器值就加1；当引用失效时，计数器值就减1；任何时刻计数器为0的对象就是不可能被再使用的。

主流的**JVM**里面没有选用引用计数算法来管理内存，其中最主要的原因是它很难解决对象间的互循环引用的问题。

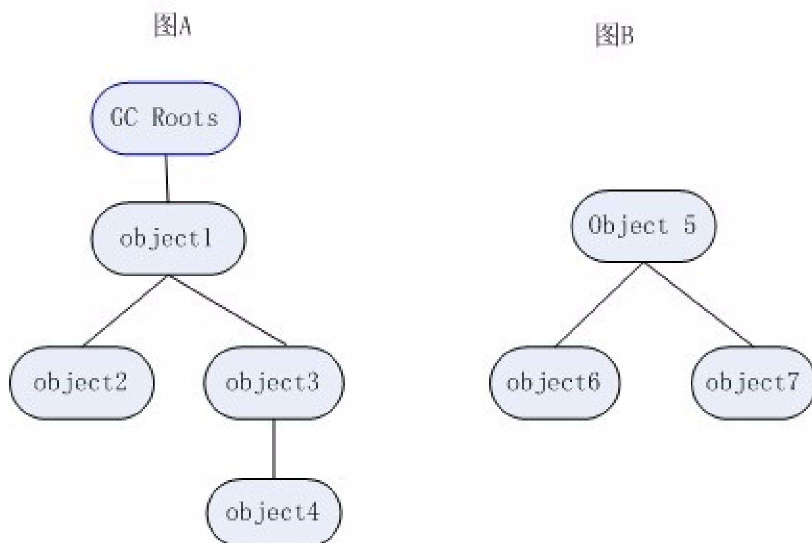
二、可达性分析算法

通过一些列的称为“GC Roots”的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链，当一个对象到GC Roots没有任何引用链相连时（就是从GC Roots 到这个对象是不可达），则证明此对象是不可用的。所以它们会被判定为可回收对象（例如图B中的对象既是不可达的）。

在Java语言中，可以作为GC Roots的对象包括下面几种：

- 虚拟机栈（栈帧中的本地变量表）中引用的对象；
- 方法区中类静态属性引用的对象；
- 方法区中常量引用的对象；
- 本地方法栈中JNI（即一般说的Native方法）引用的对象；

总结就是，方法运行时，方法中引用的对象；类的静态变量引用的对象；类中常量引用的对象；Native方法中引用的对象



在可达性分析算法中，要眞正宣告一个对象死亡，至少要经历两次标记过程：

1.如果对象在进行可达性分析后发现没有与GC Roots相连接的引用链，那它将会被第一次标记并且进行一次筛选，筛选的条件是此对象是否有必要执行finalize()方法。当对象没有 覆盖finalize()方法，或者finalize()方法已经被虚拟机调用过，虚拟机将这两种情况都视为“没有必要执行”。

2.如果这个对象被判定为有必要执行finalize()方法，那么这个对象将会放置在一个叫做F-Queue队列之中，并在稍后由一个由虚拟机自动建立的、低优先级的Finalizer线程去执行它。finalize()方法是对象逃脱死亡命运的最后一次机会，稍候GC将对F-Queue中的对象进行第二次小规模的标记，如果对象要在finalie()中成功拯救自己——只要重新与引用链上的任何一个对象建立关联即可，譬如把自己（this关键字）赋值给某个类变量或者对象的成员变量，那在第二次标记时它将会被移除出“即将回收”的集合；如果对象这时候还没有逃脱，那基本上它就眞的被回收了。

三、判断对象是否存活与“引用”有关

在JDK1.2之后，Java对引用的概念进行了扩充，将引用分为强引用（Strong Reference）、软引用（Soft Reference）、弱引用（Weak Reference）、虚引用（Phantom Reference）四种，这四种引用强度依次逐渐减弱。

强引用： 就是指在程序代码之中普遍存在的，类似“Object obj = new Object()”这类的引用，只要强引用还存在，垃圾收集器永远不会回收掉被引用的对象。

软引用： 用来描述一些还有用但并非必须的对象。在系统将要发生内存溢出异常之前，将会把这些对象列进回收范围之中进行第二次回收。

弱引用： 用户描述非必须对象的。被弱引用关联的对象只能生存到下一次垃圾收集发生之前。当垃圾收集器工作时，无论当前内存是否足够，都会回收掉只被弱引用关联的对象。

虚引用： 一个对象是否有虚引用存在，完全不会对其生存时间构成影响，也无法通过虚引用来取得一个对象实例。为一个对象设置虚引用的唯一目的就是能在这个对象被收集器回收时刻得到一个系统通知。

一、前言

本部分内容主要包含以下：

- 四大组件
- 事件分发机制
- 消息机制
- binder
- 线程与进程
- 其他面试点

以上内容都是Android中的基础知识，对于Android的学习很有帮助。其中事件分发机制、消息机制、binder等都是面试常问知识点，是必须要掌握的。

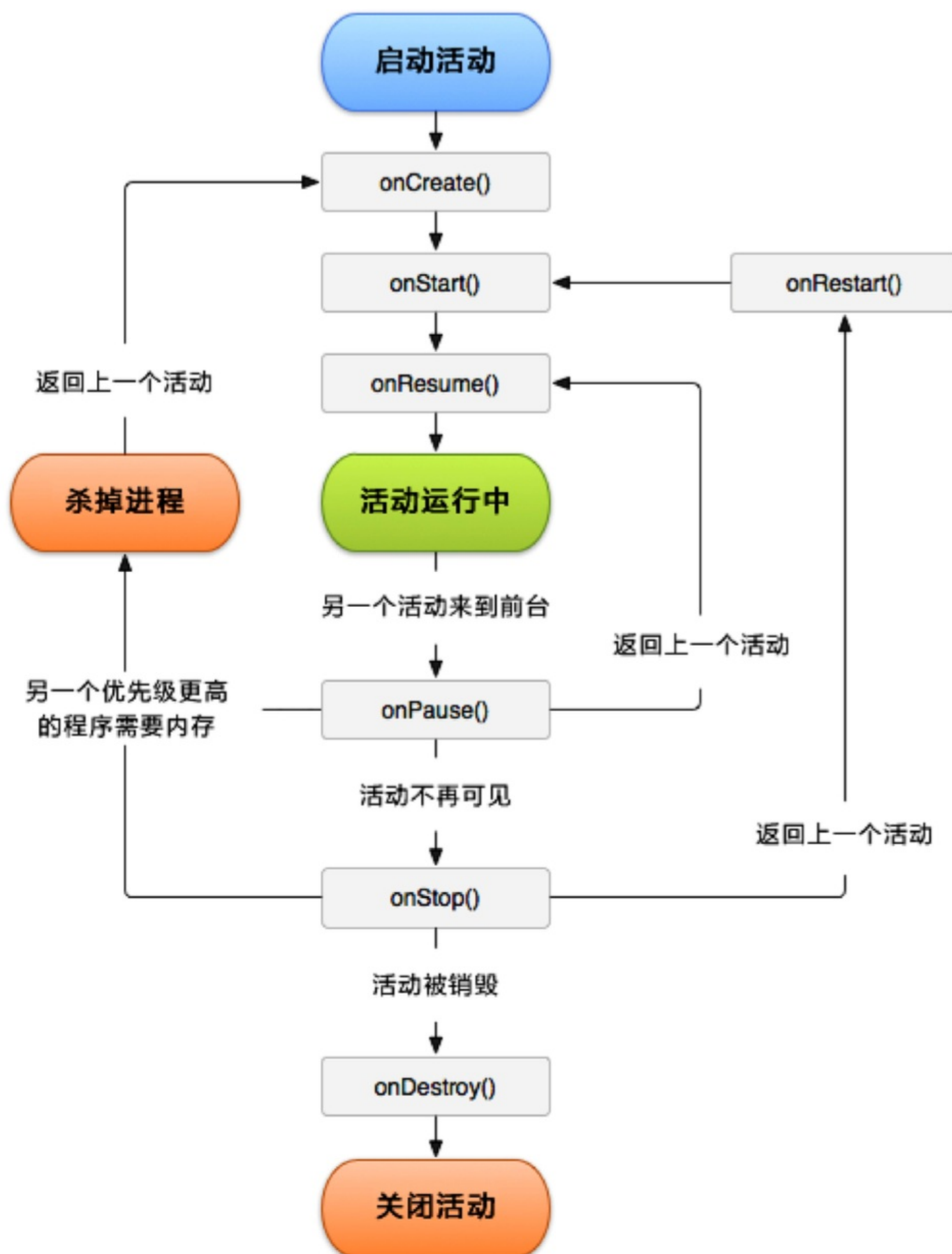
二、目录

一、Activity的生命周期

本节内容将生命周期的情况分为两部分介绍，第一部分先了解典型的生命周期的7个部分及Activity的状态。第二部分会介绍Activity在一些特殊情况下的生命周期的经历过程。

1.典型的生命周期的了解

先上经典图片。



关于这张图片，我们可能在初学Android时就有接触，今天我们继续回顾一下。

在正常情况下，一个Activity从启动到结束会以如下顺序经历整个生命周期：onCreate()->onStart()->onResume()->onPause()->onStop()->onDestory()。包含了六个部分，还有一个onRestart()没有调用，下面我们一一介绍这七部分内容。

(1) onCreate()：当 Activity 第一次创建时会被调用。这是生命周期的第一个方法。在这个方法中，可以做一些初始化工作，比如调用setContentView去加载界面布局资源，初始化Activity所需的数据。当然也可借助onCreate()方法中的Bundle对象来回复异常情况下Activity结束时的状态（后面会介绍）。

(2) onRestart()：表示Activity正在重新启动。一般情况下，当当前Activity从不可见重新变为可见状态时，onRestart就会被调用。这种情形一般是用户行为导致的，比如用户按Home键切换到桌面或打开了另一个新的Activity，接着用户又回到了这个Actvity。（关于这部分生命周期的历经过程，后面会介绍。）

(3) onStart(): 表示Activity正在被启动，即将开始，这时Activity已经出现了，但是还没有出现在前台，无法与用户交互。这个时候可以理解为**Activity**已经显示出来，但是我们还看不到。

(4) onResume():表示Activity已经可见了，并且出现在前台并开始活动。需要和onStart()对比，onStart的时候Activity还在后台，onResume的时候Activity才显示到前台。

(5) onPause():表示 Activity正在停止，仍可见，正常情况下，紧接着onStop就会被调用。在特殊情况下，如果这个时候快速地回到当前Activity，那么onResume就会被调用（极端情况）。**onPause**中不能进行耗时操作，会影响到新**Activity**的显示。因为**onPause**必须执行完，新的**Activity**的**onResume**才会执行。

(6) onStop():表示Activity即将停止，不可见，位于后台。可以做稍微重量级的回收工作，同样不能太耗时。

(7) onDestory():表示Activity即将销毁，这是Activity生命周期的最后一个回调，可以做一些回收工作和最终的资源回收。

在平常的开发中，我们经常用到的就是 onCreate()和onDestory()，做一些初始化和回收操作。

生命周期的几种普通情况

①针对一个特定的Activity，第一次启动，回调如下：onCreate()->onStart()->onResume()

②用户打开新的Activiy的时候，上述Activity的回调如下：onPause()->onStop()

③再次回到原Activity时，回调如下：onRestart()->onStart()->onResume()

④按back键回退时，回调如下：onPause()->onStop()->onDestory()

⑤按Home键切换到桌面后又回到该Actitivy，回调如下：onPause()->onStop()->onRestart()->onStart()->onResume()

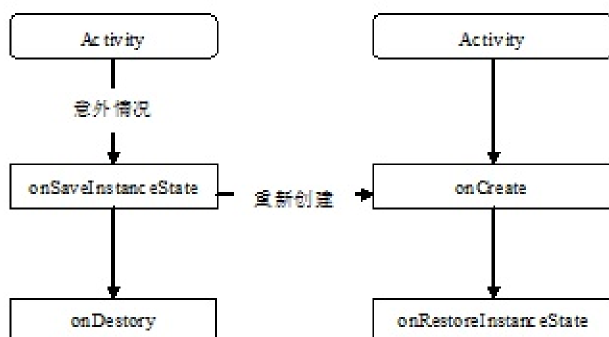
⑥调用finish()方法后，回调如下：onDestory()**(**以在**onCreate()**方法中调用为例，不同方法中回调不同，通常都是在**onCreate()**方法中调用**)**

2.特殊情况下的生命周期

上面是普通情况下Activity生命周期的一些流程，但是在一些特殊情况下，Activity的生命周期的经历有些异常，下面就是两种特殊情况。
①横竖屏切换
在横竖屏切换的过程中，会发生Activity被销毁并重建的过程。

在了解这种情况下的生命周期时，首先应该了解这两个回调：**onSaveInstanceState**和**onRestoreInstanceState**。

在Activity由于异常情况下终止时，系统会调用onSaveInstanceState来保存当前Activity的状态。这个方法的调用是在onStop之前，它和onPause没有既定的时序关系，该方法只在Activity被异常终止的情况下调用。当异常终止的Activity被重建以后，系统会调用onRestoreInstanceState，并且把Activity销毁时onSaveInstanceState方法所保存的Bundle对象参数同时传递给onRestoreInstanceState和onCreate方法。因此，可以通过onRestoreInstanceState方法来恢复Activity的状态，该方法的调用时机是在onStart之后。其中**onCreate**和**onRestoreInstanceState**方法来恢复**Activity**的状态的区别： onRestoreInstanceState回调则表明其中Bundle对象非空，不用加非空判断。onCreate需要非空判断。建议使用onRestoreInstanceState。



横竖屏切换的生命周期：onPause()->onSaveInstanceState()-> onStop()->onDestroy()->onCreate()->onStart()->onRestoreInstanceState->onResume()

可以通过在AndroidManifest文件的Activity中指定如下属性：

```
android:configChanges = "orientation| screenSize"
```

来避免横竖屏切换时，Activity的销毁和重建，而是回调了下面的方法：

```
@Override
    public void onConfigurationChanged(Configuration newConfig)
{
        super.onConfigurationChanged(newConfig);
    }
```

②资源内存不足导致优先级低的Activity被杀死
Activity优先级的划分和下面的Activity的三种运行状态是对应的。

(1) 前台Activity——正在和用户交互的Activity，优先级最高。

(2) 可见但非前台Activity——比如Activity中弹出了一个对话框，导致Activity可见但是位于后台无法和用户交互。

(3) 后台Activity——已经被暂停的Activity，比如执行了onStop，优先级最低。

当系统内存不足时，会按照上述优先级从低到高去杀死目标Activity所在的进程。我们在平常使用手机时，能经常感受到这一现象。这种情况下数组存储和恢复过程和上述情况一致，生命周期情况也一样。

3.Activity的三种运行状态

①Resumed（活动状态）

又叫Running状态，这个Activity正在屏幕上显示，并且有用户焦点。这个很好理解，就是用户正在操作的那个界面。

②Paused（暂停状态）

这是一个比较不常见的状态。这个Activity在屏幕上是可见的，但是并不是在屏幕最前端的那个Activity。比如有另一个非全屏或者透明的Activity是Resumed状态，没有完全遮盖这个Activity。

③Stopped（停止状态）

当Activity完全不可见时，此时Activity还在后台运行，仍然在内存中保留Activity的状态，并不是完全销毁。这个也很好理解，当跳转的另外一个界面，之前的界面还在后台，按回退按钮还会恢复原来的状态，大部分软件在打开的时候，直接按Home键，并不会关闭它，此时的Activity就是Stopped状态。

二、Activity的启动模式

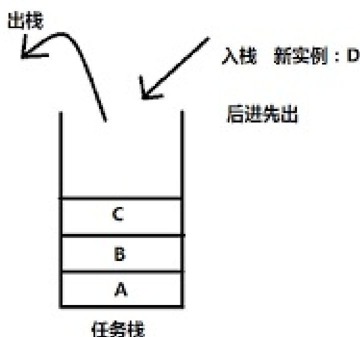
1.启动模式的类别

Android提供了四种Activity启动方式：

标准模式（standard）
栈顶复用模式（singleTop）
栈内复用模式（singleTask）
单例模式（singleInstance）

2.启动模式的结构——栈

Activity的管理是采用任务栈的形式，任务栈采用“后进先出”的栈结构。

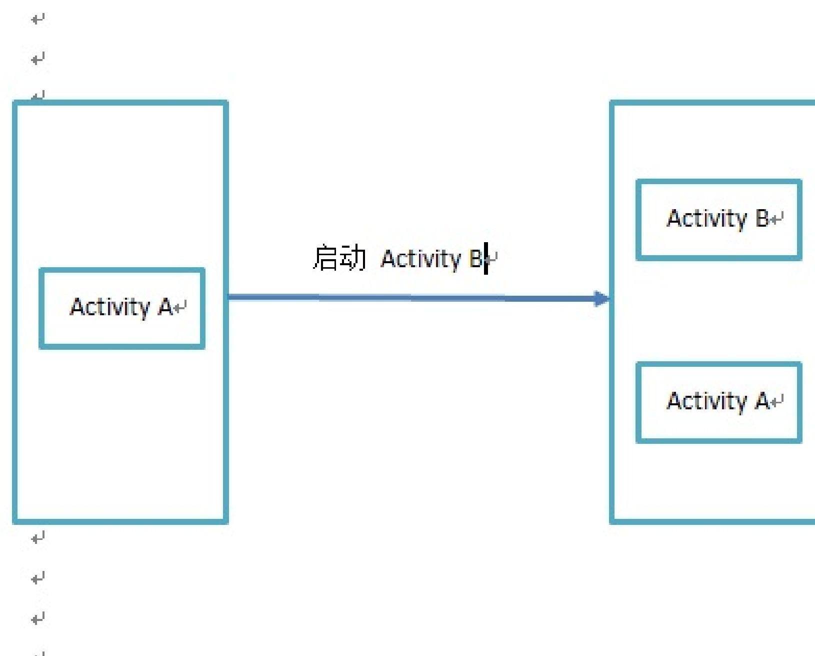


3.Activity的LaunchMode

(1)标准模式（standard）

每启动一次Activity，就会创建一个新的Activity实例并置于栈顶。谁启动了这个Activity，那么这个Activity就运行在启动它的那个Activity所在的栈中。

例如：Activity A启动了Activity B，则就会在A所在的栈顶压入一个新的Activity。



特殊情况，如果在**Service**或**Application**中启动一个**Activity**，其并没有所谓的任务栈，可以使用标记位**Flag**来解决。解决办法：为待启动的**Activity**指定**FLAG_ACTIVITY_NEW_TASK**标记位，创建一个新栈。

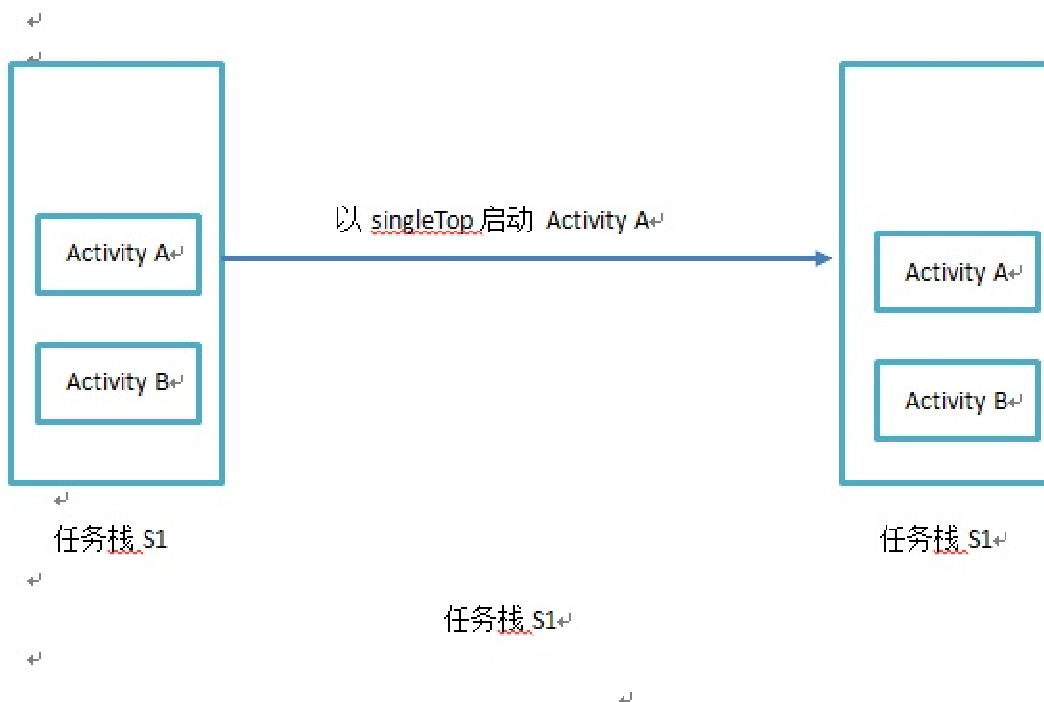
应用场景： 绝大多数Activity。如果以这种方式启动的Activity被跨进程调用，在5.0之前新启动的Activity实例会放入发送Intent的Task的栈的顶部，尽管它们属于不同的程序，这似乎有点费解看起来也不是那么合理，所以在5.0之后，上述情景会创建一个新的Task，新启动的Activity就会放入刚创建的Task中，这样就合理的多了。

(2)栈顶复用模式（**singleTop**）

如果需要新建的Activity位于任务栈栈顶，那么此Activity的实例就不会重建，而是重用栈顶的实例。并回调如下方法：

```
@Override
protected void onNewIntent(Intent intent) {
    super.onNewIntent(intent);
}
```

由于不会重建一个Activity实例，则不会回调其他生命周期方法。
如果栈顶不是新建的Activity,就会创建该Activity新的实例，并放入栈顶。



应用场景： 在通知栏点击收到的通知，然后需要启动一个Activity，这个Activity就可以用singleTop，否则每次点击都会新建一个Activity。当然实际的开发过程中，测试妹纸没准给你提过这样的bug：某个场景下连续快速点击，启动了两个Activity。如果这个时候待启动的Activity使用 singleTop模式也是可以避免这个Bug的。同standard模式，如果是外部程序启动singleTop的Activity，在Android 5.0之前新创建的Activity会位于调用者的Task中，5.0及以后会放入新的Task中。

(3)栈内复用模式（**singleTask**）

该模式是一种单例模式，即一个栈内只有一个该Activity实例。该模式，可以通过在AndroidManifest文件的Activity中指定该Activity需要加载到那个栈中，即singleTask的Activity可以指定想要加载的目标栈。singleTask和taskAffinity配合使用，指定开启的Activity加入到哪个栈中。

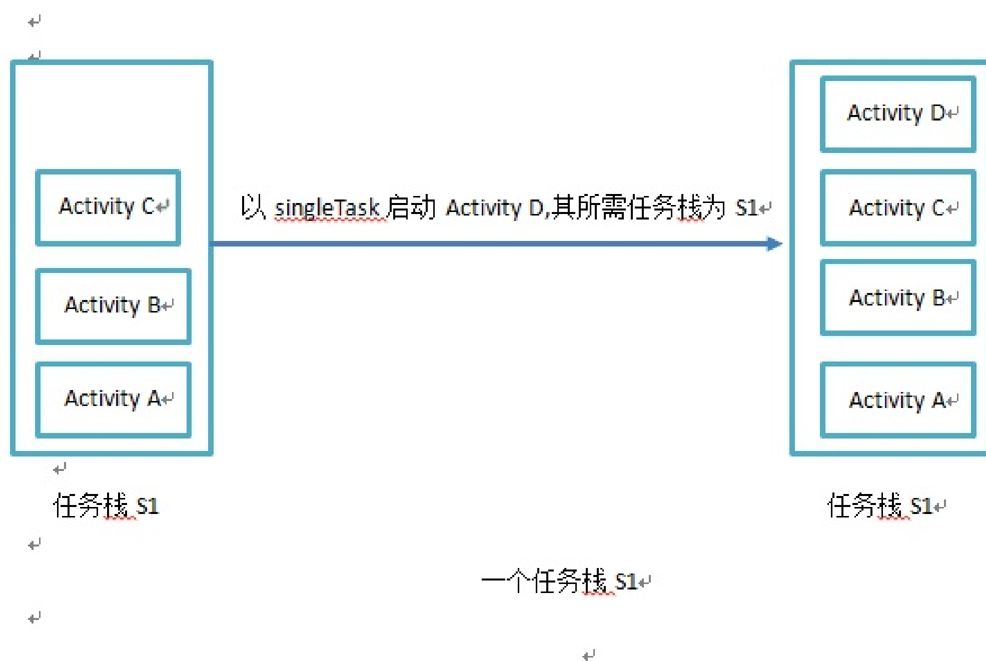
```
<activity android:name=".Activity1"
    android:launchMode="singleTask"
    android:taskAffinity="com.lvr.task"
    android:label="@string/app_name">
</activity>
```

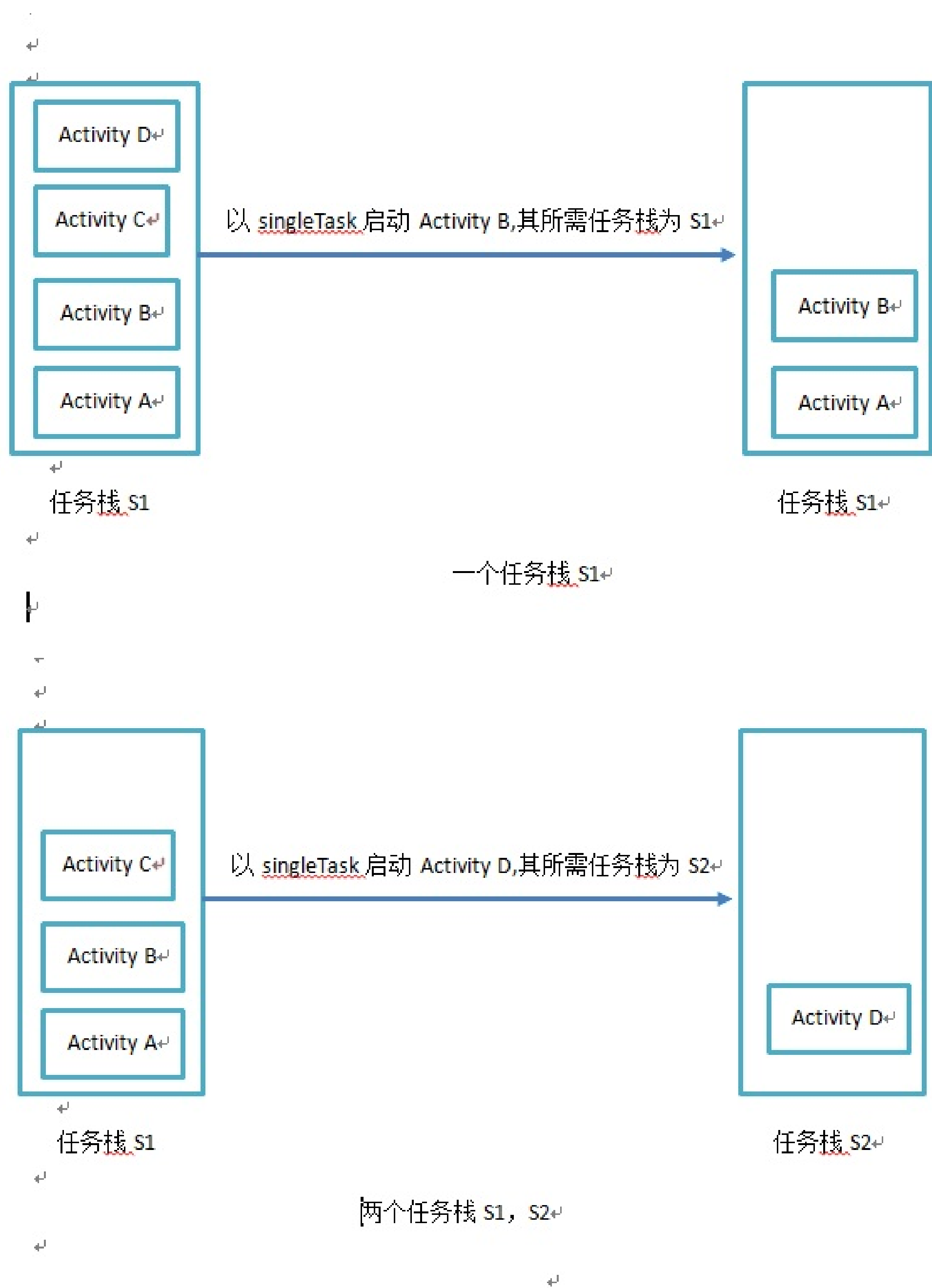
关于**taskAffinity**的值： 每个Activity都有taskAffinity属性，这个属性指出了它希望进入的Task。如果一个Activity没有显式的指明该Activity的taskAffinity，那么它的这个属性就等于Application指明的taskAffinity，如果Application也没有指明，那么该taskAffinity的值就等于包名。

执行逻辑：

在这种模式下，如果Activity指定的栈不存在，则创建一个栈，并把创建的Activity压入栈内。如果Activity指定的栈存在，如果其中没有该Activity实例，则会创建Activity并压入栈顶，如果其中有该Activity实例，则把该Activity实例之上的Activity杀死清除出栈，重用并让该Activity实例处在栈顶，然后调用onNewIntent()方法。

对应如下三种情况：

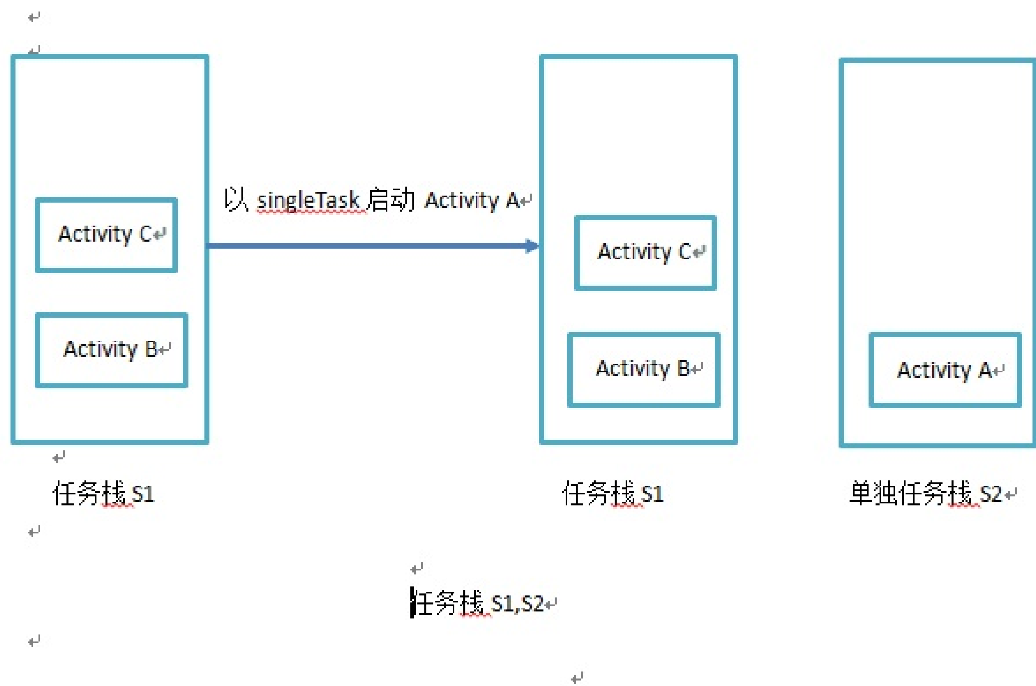




应用场景： 大多数App的主页。对于大部分应用，当我们在主界面点击回退按钮的时候都是退出应用，那么当我们第一次进入主界面之后，主界面位于栈底，以后不管我们打开了多少个Activity，只要我们再次回到主界面，都应该使用将主界面Activity上所有的Activity移除的方式来让主界面Activity处于栈顶，而不是往栈顶新加一个主界面Activity的实例，通过这种方式能够保证退出应用时所有的Activity都能报销毁。在跨应用Intent传递时，如果系统中不存在singleTask Activity的实例，那么将创建一个新的Task，然后创建SingleTask Activity的实例，将其放入新的Task中。

(4)单例模式（singleInstance）

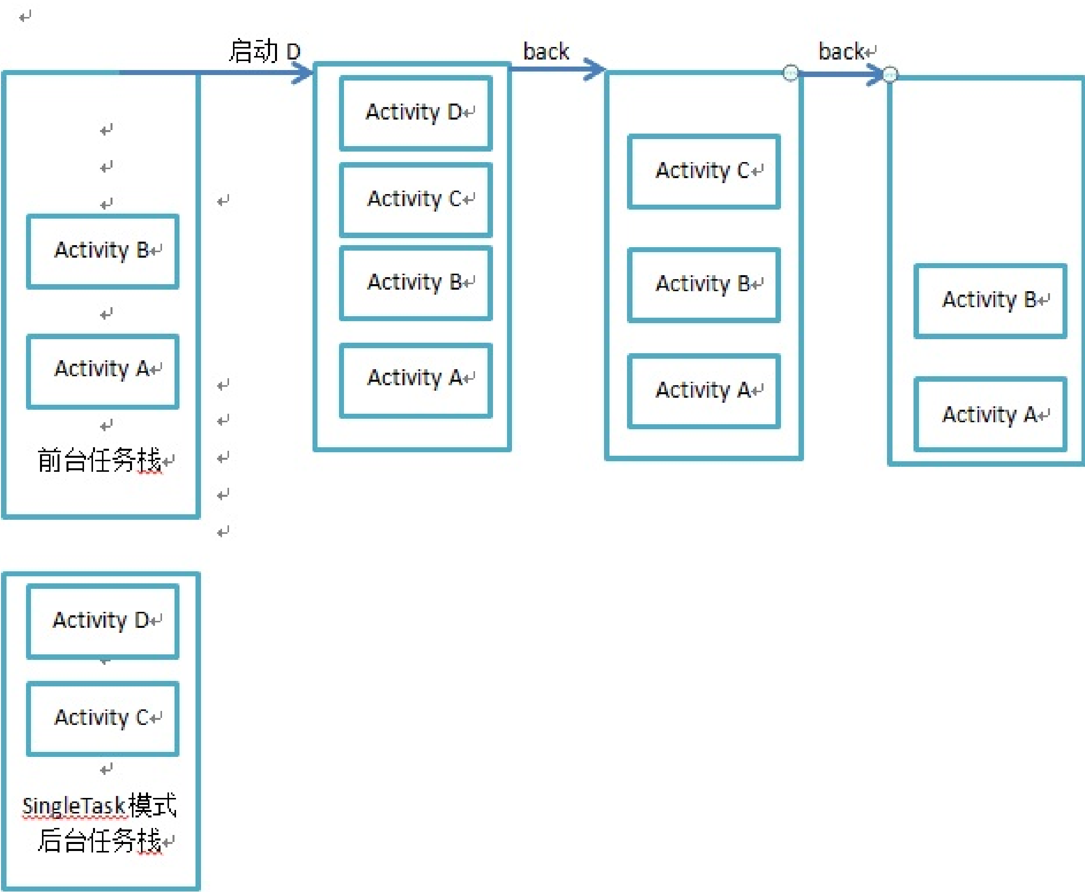
作为栈内复用模式（singleTask）的加强版,打开该Activity时，直接创建一个新的任务栈，并创建该Activity实例放入新栈中。一旦该模式的Activity实例已经存在于某个栈中，任何应用再激活该Activity时都会重用该栈中的实例。



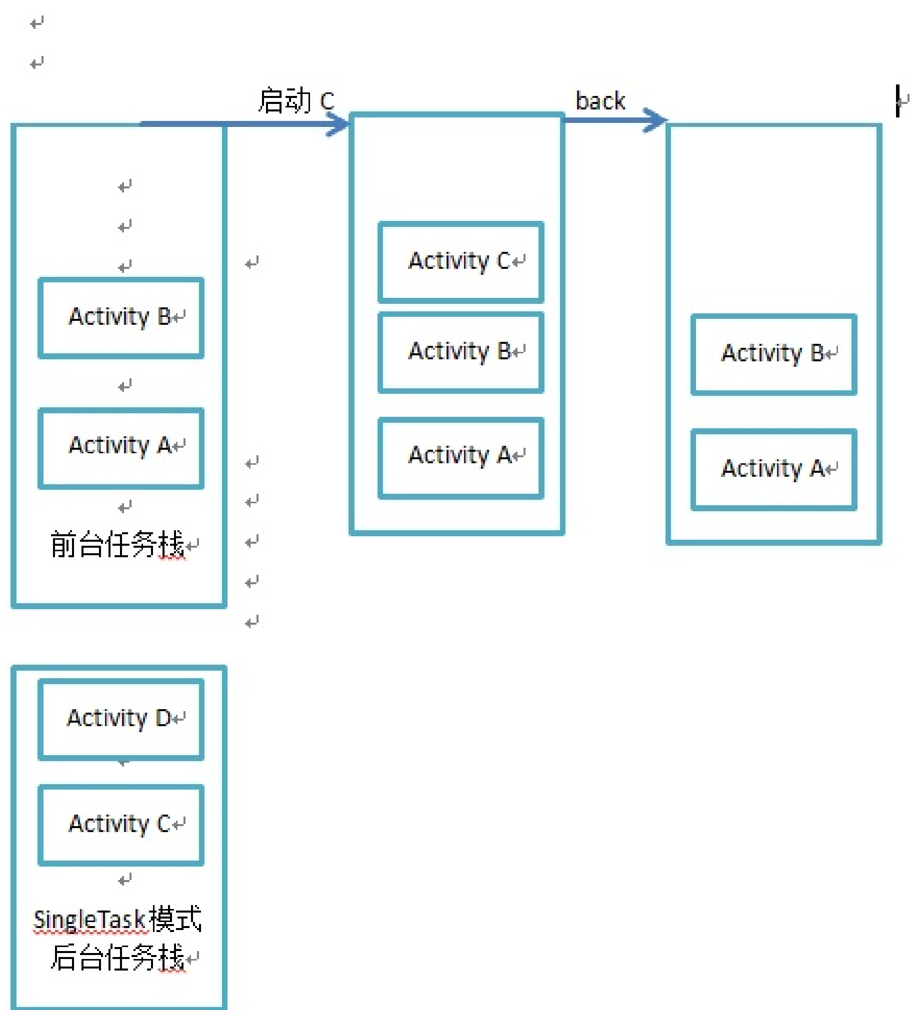
应用场景：呼叫来电界面。这种模式的使用情况比较罕见，在Launcher中可能使用。或者你确定你需要使Activity只有一个实例。建议谨慎使用。

3.特殊情况——前台栈和后台栈的交互

假如目前有两个任务栈。前台任务栈为AB，后台任务栈为CD，这里假设CD的启动模式均为singleTask,现在请求启动D，那么这个后台的任务栈都会被切换到前台，这个时候整个后退列表就变成了ABCD。当用户按back返回时，列表中的activity会一一出栈，如下图。



如果不是请求启动D而是启动C，那么情况又不一样，如下图。



调用**SingleTask**模式的后台任务栈中的**Activity**，会把整个栈的**Actvity**压入当前栈的栈顶。**singleTask**会具有**clearTop**特性，把之上的栈内**Activity**清除。

4.Activity的Flags

Activity的Flags很多，这里介绍集中常用的，用于设定Activity的启动模式。可以在启动Activity时，通过Intent的addFlags()方法设置。

(1)FLAG_ACTIVITY_NEW_TASK 其效果与指定Activity为singleTask模式一致。

(2)FLAG_ACTIVITY_SINGLE_TOP 其效果与指定Activity为singleTop模式一致。

(3)FLAG_ACTIVITY_CLEAR_TOP 具有此标记位的Activity，当它启动时，在同一个任务栈中所有位于它上面的Activity都要出栈。如果和singleTask模式一起出现，若被启动的Activity已经存在栈中，则清除其之上的Activity，并调用该Activity的onNewIntent方法。如果被启动的Activity采用standard模式，那么该Activity连同之上的所有Activity出栈，然后创建新的Activity实例并压入栈中。

时间： 2018-05-12 01:44:07

一、Service简介

Service是Android程序中四大基础组件之一，它和Activity一样都是Context的子类，只不过它没有UI界面，是在后台运行的组件。

Service是Android中实现程序后台运行的解决方案，它非常适用于去执行那些不需要和用户交互而且还要求长期运行的任务。Service默认并不会运行在子线程中，它也不运行在一个独立的进程中，它同样执行在UI线程中，因此，不要在Service中执行耗时的操作，除非你在Service中创建了子线程来完成耗时操作。

二、Service种类

按运行地点分类：

类别	区别	优点	缺点	应用
本地服务(Local Service)	该服务依附在主进程上	服务依附在主进程上而不是独立的进程，这样在一定程度上节约了资源，另外Local服务因为是在同一进程因此不需要IPC，也不需要AIDL。相应`bindService`会方便很多。	主进程被Kill后，服务便会终止。	如：音乐播放器播放等不需要常驻的服务。
远程服务(Remote Service)	该服务是独立的进程	服务为独立的进程，对应进程名格式为所在包名加上你指定的`android:process`字符串。由于是独立的进程，因此在Activity所在进程被Kill的时候，该服务依然在运行，不受其他进程影响，有利于为多个进程提供服务具有较高的灵活性。	该服务是独立的进程，会占用一定资源，并且使用AIDL进行IPC稍微麻烦一点。	一些提供系统服务的Service，这种Service是常驻的。

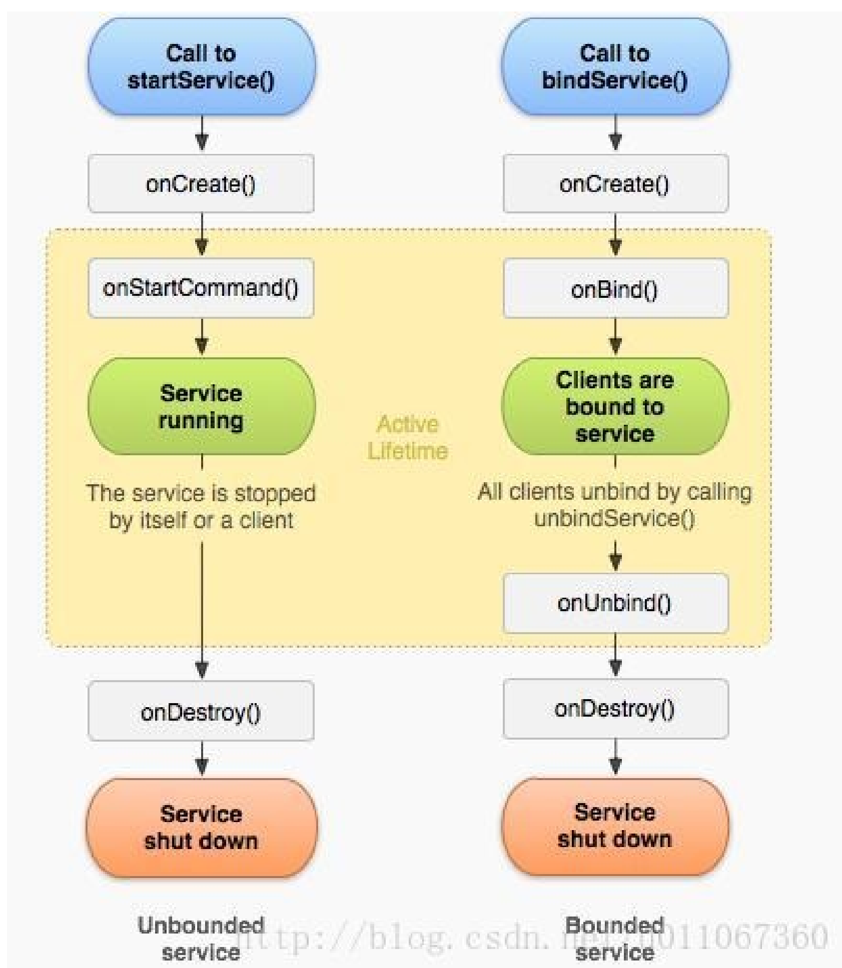
按运行类型分类：

类别	区别	应用
前台服务	会在通知栏显示onGoing的Notification	当服务被终止的时候，通知一栏的Notification 也会消失，这样对于用户有一定的通知作用。常见的如音乐播放服务。
后台服务	默认的服务即为后台服务，即不会在通知一栏显示 onGoing 的 Notification。	当服务被终止的时候，用户是看不到效果的。某些不需要运行或终止提示的服务，如天气更新，日期同步，邮件同步等。

按使用方式分类：

类别	区别
startService启动的服务	主要用于启动一个服务执行后台任务，不进行通信。停止服务使用stopService。
bindService启动的服务	方法启动的服务要进行通信。停止服务使用unbindService。
同时使用startService、bindService 启动的服务	停止服务应同时使用stopService与unbindService。

三、Service生命周期



OnCreate()

系统在service第一次创建时执行此方法，来执行只运行一次的初始化工作。如果service已经运行，这个方法不会被调用。

onStartCommand()

每次客户端调用startService()方法启动该Service都会回调该方法（多次调用）。一旦这个方法执行，service就启动并且在后台长期运行。通过调用stopSelf()或stopService()来停止服务。

OnBind()

当组件调用bindService()想要绑定到service时(比如想要执行进程间通讯)系统调用此方法（一次调用，一旦绑定后，下次再调用bindService()不会回调该方法）。在你的实现中，你必须提供一个返回一个IBinder来以使客户端能够使用它与service通讯，你必须总是实现这个方法，但是如果你不允许绑定，那么你应返回null。

OnUnbind()

当前组件调用unbindService()，想要解除与service的绑定时系统调用此方法（一次调用，一旦解除绑定后，下次再调用unbindService()会抛出异常）。

OnDestory()
系统在service不再被使用并要销毁时调用此方法（一次调用）。service应在此方法中释放资源，比如线程，已注册的侦听器，接收器等等．这是service收到的最后一个调用。

下面介绍三种不同情况下Service的生命周期情况。

1.startService / stopService

生命周期顺序：onCreate->onStartCommand->onDestroy

如果一个Service被某个Activity 调用 Context.startService方法启动，那么不管是否有Activity使用bindService绑定或unbindService解除绑定到该Service，该Service都在后台运行，直到被调用stopService，或自身的stopSelf方法。当然如果系统资源不足，android系统也可能结束服务，还有一种方法可以关闭服务，在设置中，通过应用**->**找到自己应用**->**停止。

注意点：

①第一次 startService 会触发 onCreate 和 onStartCommand，以后在服务运行过程中，每次 startService 都只会触发 onStartCommand

②不论 startService 多少次，stopService 一次就会停止服务

2.bindService / unbindService

生命周期顺序：onCreate->onBind->onUnBind->onDestroy

如果一个Service在某个Activity中被调用bindService方法启动，不论bindService被调用几次，Service的onCreate方法只会执行一次，同时onStartCommand方法始终不会调用。

当建立连接后，Service会一直运行，除非调用unbindService来接触绑定、断开连接或调用该Service的Context不存在了（如Activity被Finish——即通过**bindService**启动的**Service**的生命周期依附于启动它的**Context**），系统在这时会自动停止该Service。

注意点：

第一次 bindService 会触发 onCreate 和 onBind，以后在服务运行过程中，每次 bindService 都不会触发任何回调

3.混合型（上面两种方式的交互）

当一个Service在被启动(startService)的同时又被绑定(bindService)，该Service将会一直在后台运行，并且不管调用几次，onCreate方法始终只会调用一次，onStartCommand的调用次数与startService调用的次数一致（使用bindService方法不会调用onStartCommand）。同时，调用**unBindService**将不会停止**Service**，必须调用**stopService**或**Service**自身的**stopSelf**来停止服务。

在什么情况下使用 **startService** 或 **bindService** 或 同时使用 **startService** 和 **bindService**？

①如果你只是想要启动一个后台服务长期进行某项任务那么使用 startService 便可以了。

②如果你想要与正在运行的 Service 取得联系，那么有两种方法，一种是使用 broadcast ，另外是使用 bindService ，前者的缺点是如果交流较为频繁，容易造成性能上的问题，并且 BroadcastReceiver 本身执行代码的时间是很短的（也许执行到一半，后面的代码便不会执行），而后者则没有这些问题，因此我们肯定选择使用 bindService（这个时候你便同时在使用 startService 和 bindService 了，这在 Activity 中更新 Service 的某些运行状态是相当有用的）。

③如果你的服务只是公开一个远程接口，供连接上的客服端（android 的 Service 是C/S架构）远程调用执行方法。这个时候你可以不让服务一开始就运行，而只用 bindService ，这样在第一次 bindService 的时候才会创建服务的实例运行它，这会节约很多系统资源，特别是如果你的服务是Remote Service，那么该效果会越明显（当然在 Service 创建的时候会花去一定时间，你应当注意到这点）。

四、Service的几种典型使用实例

1.不可交互的后台服务

不可交互的后台服务即是普通的Service，通过startService()方式开启。Service的生命周期很简单，分别为onCreate、onStartCommand、onDestroy这三个。

创建服务类：

```
public class BackService extends Service {
    private Thread mThread;
```

```
    @Override
    public void onCreate() {
        super.onCreate();

    }

    @Nullable
    @Override
    public IBinder onBind(Intent intent) {
        System.out.println("onBind");
        return null;
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int star
tId) {
        //执行耗时操作

        mThread = new Thread() {
            @Override
            public void run() {
                try {
                    while (true) {
                        //等待停止线程
                        if (this.isInterrupted()) {
                            throw new InterruptedException();
                        }
                        //耗时操作。
                        System.out.println("执行耗时操作");
                    }
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        };
        mThread.start();
        return super.onStartCommand(intent, flags, startId);
    }
```

```
    @Override
    public void onDestroy() {
        super.onDestroy();
        //停止线程
        mThread.interrupt();
    }
}
```

配置服务：

```
<service android:name=".BackService">
</service>
```

如果想配置成远程服务，加如下代码：

```
android:process="remote"
```

配置好Service类，只需要在前台，调用startService()方法，就会启动耗时操作。

注意：

①不运行在一个独立的进程中，它同样执行在UI线程中，因此，在Service中创建了子线程来完成耗时操作。

②当Service关闭后，如果在onDestory()方法中不关闭线程，你会发现我们的子线程进行的耗时操作是一直存在的，此时关闭该子线程的方法需要直接关闭该应用程序。因此，在**onDestory()**方法中要进行必要的清理工作。

2.可交互的后台服务

可交互的后台服务是指前台页面可以调用后台服务的方法，通过bindService()方式开启。Service的生命周期很简单，分别为onCreate、onBind、onUnBind、onDestroy这四个。
可交互的后台服务实现步骤是和不可交互的后台服务实现步骤是一样的，区别在于启动的方式和获得Service的代理对象。

创建服务类
和普通Service不同在于这里返回一个代理对象，返回给前台进行获取，即前台可以获取该代理对象执行后台服务的方法

```
public class BackService extends Service {

    @Override
    public void onCreate() {
        super.onCreate();

    }

    @Nullable
    @Override
    public IBinder onBind(Intent intent) {
        //返回MyBinder对象
        return new MyBinder();
    }
    //需要返回给前台的Binder类
    class MyBinder extends Binder {
        public void showTip(){
            System.out.println("我是来此服务的提示");
        }
    }
    @Override
    public void onDestroy() {
        super.onDestroy();

    }
}
```

前台调用
通过以下方式绑定服务：

```
bindService(mIntent,con,BIND_AUTO_CREATE);
```

其中第二个参数：

```
private ServiceConnection con = new ServiceConnection() {
        @Override
        public void onServiceConnected(ComponentName name, IBind
er service) {
            BackService.MyBinder myBinder = (BackService.MyBinde
r) service;
            myBinder.showTip();
        }

        @Override
        public void onServiceDisconnected(ComponentName name) {

        }
    };
```

当建立绑定后，onServiceConnected中的service便是Service类中onBind的返回值。如此便可以调用后台服务类的方法，实现交互。

当调用unbindService()停止服务，同时要在onDestory()方法中做好清理工作。

注意：通过**bindService**启动的**Service**的生命周期依附于启动它的**Context**。因此当前台调用**bindService**的**Context**销毁后，那么服务会自动停止。

3.混合型后台服务

将上面两种启动方式结合起来就是混合性交互的后台服务了，即可以单独运行后台服务，也可以运行后台服务中提供的方法，其完整的生命周期是：onCreate->onStartCommand->onBind->onUnBind->onDestroy

4.前台服务

所谓前台服务只不是通过一定的方式将服务所在的进程级别提升了。前台服务会一直有一个正在运行的图标在系统的状态栏显示，非常类似于通知的效果。

由于后台服务优先级相对比较低，当系统出现内存不足的情况下，它就有可能会被回收掉，所以前台服务就是来弥补这个缺点的，它可以一直保持运行状态而不被系统回收。

创建服务类

前台服务创建很简单，其实就在Service的基础上创建一个Notification，然后使用Service的startForeground()方法即可启动为前台服务。

```
public class ForeService extends Service{
    @Nullable
    @Override
    public IBinder onBind(Intent intent) {
        return null;
    }

    @Override
    public void onCreate() {
        super.onCreate();
        beginForeService();
    }

    private void beginForeService() {
        //创建通知
        Notification.Builder mBuilder = new Notification.Builder(
this)
                .setSmallIcon(R.mipmap.ic_launcher)
                .setContentText("2017-2-27")
                .setContentText("您有一条未读短信...");
        //创建点跳转的Intent(这个跳转是跳转到通知详情页)
        Intent intent = new Intent(this,NotificationShow.class);
        //创建通知详情页的栈
        TaskStackBuilder stackBulider = TaskStackBuilder.create(
this);
        //为其添加父栈 当从通知详情页回退时，将退到添加的父栈中
        stackBulider.addParentStack(NotificationShow.class);
        PendingIntent pendingIntent = stackBulider.getPendingInt
ent(0,PendingIntent.FLAG_UPDATE_CURRENT);
        //设置跳转Intent到通知中
        mBuilder.setContentIntent(pendingIntent);
        //获取通知服务
        NotificationManager nm = (NotificationManager) getSystem
Service(Context.NOTIFICATION_SERVICE);
        //构建通知
        Notification notification = mBuilder.build();
        //显示通知
```

```
            nm.notify(0,notification);
            //启动前台服务
            startForeground(0,notification);
        }
    }
```

启动前台服务

```
startService(new Intent(this, ForeService.class));
```

关于TaskStackBuilder 这一段，可能不是看的很明白，下面详细介绍。

TaskStackBuilder在Notification通知栏中的使用

首先是用一般的PendingIntent来进行跳转

```
mBuilder = new NotificationCompat.Builder(this).setContent(view)
        .setSmallIcon(R.drawable.icon).setTicker("新资讯")
        .setWhen(System.currentTimeMillis())
        .setOngoing(false)
        .setAutoCancel(true);
Intent intent = new Intent(this, NotificationShow.class);
PendingIntent pendingIntent = PendingIntent.getActivity(this, 0,
 intent, PendingIntent.FLAG_UPDATE_CURRENT);
mBuilder.setContentIntent(pendingIntent);
```

这里是直接用PendingIntent来跳转到NotificationShow。

在运行效果上来看，首先发送了一条Notification到通知栏上，然后这时，退出程序，即MainActivity已经不存在了，回到home主菜单，看到Notification仍然存在，当然，我们还没有点击或者cancel它，现在去点击Notification，跳转到NotificationShow界面，然后我们按下Back键，发现直接回到home菜单了。现在大多数android应用都是在通知栏中如果有Notification通知的话，点击它，然后会直接跳转到对应的应用程序的某个界面，这时如果回退，即按下Back键，会返回到该应用程序的主界面，而不是系统的home菜单。所以用上面这种PendingIntent的做法达不到目的。这里我们使用TaskStackBuilder来做。

```
mBuilder = new NotificationCompat.Builder(this)
        .setContent(view)
        .setSmallIcon(R.drawable.icon).setTicker("新资讯")
        .setWhen(System.currentTimeMillis())
        .setOngoing(false)
        .setAutoCancel(true);
Intent intent = new Intent(this, NotificationShow.class);
TaskStackBuilder stackBuilder = TaskStackBuilder.create(this);
stackBuilder.addParentStack(NotificationShow.class);
stackBuilder.addNextIntent(intent);
PendingIntent pendingIntent = stackBuilder.getPendingIntent(0, P
endingIntent.FLAG_UPDATE_CURRENT);
//PendingIntent pendingIntent = PendingIntent.getActivity(this,
0, intent, PendingIntent.FLAG_UPDATE_CURRENT);
mBuilder.setContentIntent(pendingIntent);
```

显示用TaskStackBuilder.create(this)创建一个stackBuilder实例，接下来addParentStack();

关于这个方法，我们查一下官方API文档：

> Add the activity parent chain as specified by the parentActivityName attribute of the activity (or activity-alias) element in the application's manifest to the task stack builder

这句话意思是：为跳转后的activity添加一个父activity，在activity中的manifest中添加parentActivityName即可。

那么我们就在manifest文件中添加这个属性

```
<activity
    android:name="com.lvr.service.NotificationShow"
    android:parentActivityName=".MainActivity" >
</activity>
```

这里我让它的parentActivity为MainActivity，也就是说在NotificationShow这个界面点击回退时，会跳转到MainActivity这个界面，而不是像上面一样直接回到了home菜单。

注意：通过 **stopForeground()**方法可以取消通知，即将前台服务降为后台服务。此时服务依然没有停止。通过**stopService()**可以把前台服务停止。

以上是关于Service的内容。

一、定义

- `BroadcastReceiver` （广播接收器），属于Android四大组件之一
- 在Android开发中，BroadcastReceiver的应用场景非常多广播，是一个全局的监听器，属于 `Android` 四大组件之一

> `Android` 广播分为两个角色：广播发送者、广播接收者

二、作用

- 用于监听 / 接收 应用发出的广播消息，并做出响应
- 应用场景 a. 不同组件之间通信（包括应用内 / 不同应用之间）

  b. 与 `Android` 系统在特定情况下的通信

  > 如当电话呼入时、网络可用时

  c. 多线程通信

三、实现原理

- `Android` 中的广播使用了设计模式中的观察者模式：基于消息的发布/订阅事件模型。

  > 因此，Android将广播的发送者 和 接收者 解耦，使得系统方便集成，更易扩展

- 模型中有3个角色：

  1. 消息订阅者（广播接收者）
  2. 消息发布者（广播发布者）
  3. 消息中心（ `AMS` ，即 `Activity Manager Service` ）



- 原理描述：

  1. 广播接收者 通过 `Binder` 机制在 `AMS` 注册
  2. 广播发送者 通过 `Binder` 机制向 `AMS` 发送广播
  3. `AMS` 根据 广播发送者 要求，在已注册列表中，寻找合适的广播接收者

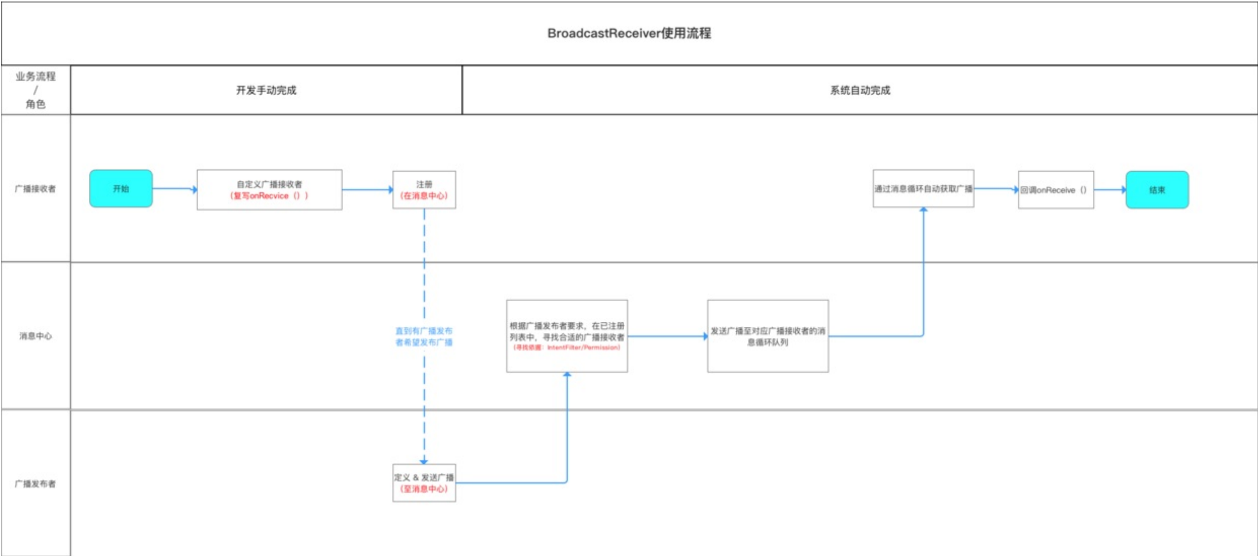
     > 寻找依据：`IntentFilter / Permission`

  4. `AMS` 将广播发送到合适的广播接收者相应的消息循环队列中；
  5. 广播接收者通过 消息循环 拿到此广播，并回调 `onReceive()`

特别注意：广播发送者 和 广播接收者的执行 是 异步的，发出去的广播不会关心有无接收者接收，也不确定接收者到底是何时才能接收到；

四、具体使用

具体使用流程如下：



接下来我将一步步介绍如何使用

> 即上图中的 开发者手动完成部分

4.1 自定义广播接收者BroadcastReceiver

- 继承自BroadcastReceivre基类

- 必须复写抽象方法onReceive()方法

> 1. 广播接收器接收到相应广播后，会自动回调onReceive()方法
> 2. 一般情况下，onReceive方法会涉及与其他组件之间的交互，如发送Notification、启动service等
> 3. 默认情况下，广播接收器运行在UI线程，因此，onReceive方法不能执行耗时操作，否则将导致ANR。

- 代码范例 *mBroadcastReceiver.java*

```
public class mBroadcastReceiver extends BroadcastReceiver {

  //接收到广播后自动调用该方法
  @Override
  public void onReceive(Context context, Intent intent) {
    //写入接收广播后的操作
    }
}
```

4.2 广播接收器注册

注册的方式分为两种：静态注册、动态注册

4.2.1 静态注册

- 在AndroidManifest.xml里通过 标签声明
- 属性说明：

```
<receiver
  android:enabled=["true" | "false"]
  //此broadcastReceiver能否接收其他App的发出的广播
  //默认值是由receiver中有无intent-filter决定的：如果有intent-filter
，默认值为true，否则为false
  android:exported=["true" | "false"]
  android:icon="drawable resource"
  android:label="string resource"
  //继承BroadcastReceiver子类的类名
  android:name=".mBroadcastReceiver"
  //具有相应权限的广播发送者发送的广播才能被此BroadcastReceiver所接收；
  android:permission="string"
  //BroadcastReceiver运行所处的进程
  //默认为app的进程，可以指定独立的进程
  //注：Android四大基本组件都可以通过此属性指定自己的独立进程
  android:process="string" >

  //用于指定此广播接收器将接收的广播类型
  //本示例中给出的是用于接收网络状态改变时发出的广播
  <intent-filter>
    <action android:name="android.net.conn.CONNECTIVITY_CHANGE"
/>
  </intent-filter>
</receiver>
```

- 注册示例

```
<receiver
  //此广播接收者类是mBroadcastReceiver
  android:name=".mBroadcastReceiver" >
  //用于接收网络状态改变时发出的广播
  <intent-filter>
      <action android:name="android.net.conn.CONNECTIVITY_CH
ANGE" />
  </intent-filter>
</receiver>
```

当此App首次启动时，系统会自动实例化mBroadcastReceiver类，并注册到系统中。

4.2.2 动态注册

在代码中通过调用Context的*registerReceiver*（）方法进行动态注册BroadcastReceiver，具体代码如下：

```
@Override
protected void onResume() {
    super.onResume();

    //实例化BroadcastReceiver子类 &  IntentFilter
    mBroadcastReceiver mBroadcastReceiver = new mBroadcastReceiv
er();
    IntentFilter intentFilter = new IntentFilter();

    //设置接收广播的类型
    intentFilter.addAction(android.net.conn.CONNECTIVITY_CHANGE)
;

    //调用Context的registerReceiver（）方法进行动态注册
    registerReceiver(mBroadcastReceiver, intentFilter);
}


//注册广播后，要在相应位置记得销毁广播
//即在onPause() 中unregisterReceiver(mBroadcastReceiver)
//当此Activity实例化时，会动态将MyBroadcastReceiver注册到系统中
//当此Activity销毁时，动态注册的MyBroadcastReceiver将不再接收到相应的广
播。
@Override
protected void onPause() {
    super.onPause();
    //销毁在onResume()方法中的广播
    unregisterReceiver(mBroadcastReceiver);
}
```

特别注意

- 动态广播最好在Activity的onResume()注册、onPause()注销。
- 原因：

  对于动态广播，有注册就必然得有注销，否则会导致内存泄露

  > 重复注册、重复注销也不允许

4.2.3 两种注册方式的区别

注册方式	特点	应用场景
静态注册 (常驻广播)	常驻，不受任何组件的生命周期影响 (应用程序关闭后，如果有信息广播来，程序依旧会被系统调用) 缺点：耗电、占内存	需要时刻监听广播
动态注册 (非常驻广播)	非常驻，灵活，跟随组件的生命周期变化 (组件结束=广播结束，在组件结束前，必须移除广播接收器)	需要特定时刻监听广播

4.3 广播发送者向AMS发送广播

4.3.1 广播的发送

- 广播是用"意图（Intent）"标识
- 定义广播的本质：定义广播所具备的"意图（Intent）"
- 广播发送：广播发送者将此广播的"意图"通过**sendBroadcast()**方法发送出去

4.3.2 广播的类型

广播的类型主要分为5类：

- 普通广播（Normal Broadcast）
- 系统广播（System Broadcast）
- 有序广播（Ordered Broadcast）
- 粘性广播（Sticky Broadcast）
- App应用内广播（Local Broadcast）

具体说明如下：

①. 普通广播（**Normal Broadcast**）

即开发者自身定义intent的广播（最常用）。发送广播使用如下：

```
Intent intent = new Intent();
//对应BroadcastReceiver中intentFilter的action
intent.setAction(BROADCAST_ACTION);
//发送广播
sendBroadcast(intent);
```

- 若被注册了的广播接收者中注册时intentFilter的action与上述匹配，则会接收此广播（即进行回调onReceive()）。如下mBroadcastReceiver则会接收上述广播

```
<receiver
    //此广播接收者类是mBroadcastReceiver
    android:name=".mBroadcastReceiver" >
    //用于接收网络状态改变时发出的广播
    <intent-filter>
        <action android:name="BROADCAST_ACTION" />
    </intent-filter>
</receiver>
```

- 若发送广播有相应权限，那么广播接收者也需要相应权限

②. 系统广播（**System Broadcast**）

- Android中内置了多个系统广播：只要涉及到手机的基本操作（如开机、网络状态变化、拍照等等），都会发出相应的广播
- 每个广播都有特定的Intent - Filter（包括具体的action），Android系统广播action如下：

系统操作	action
监听网络变化	android.net.conn.CONNECTIVITY_CHANGE
关闭或打开飞行模式	Intent.ACTION_AIRPLANE_MODE_CHANGED
充电时或电量发生变化	Intent.ACTION_BATTERY_CHANGED
电池电量低	Intent.ACTION_BATTERY_LOW
电池电量充足（即从电量低变化到饱满时会发出广播	Intent.ACTION_BATTERY_OKAY
系统启动完成后(仅广播一次)	Intent.ACTION_BOOT_COMPLETED
按下照相时的拍照按键(硬件按键)时	Intent.ACTION_CAMERA_BUTTON
屏幕锁屏	Intent.ACTION_CLOSE_SYSTEM_DIALOGS
设备当前设置被改变时(界面语言、设备方向等)	Intent.ACTION_CONFIGURATION_CHANGED
插入耳机时	Intent.ACTION_HEADSET_PLUG
未正确移除SD卡但已取出来时(正确移除方法:设置--SD卡和设备内存--卸载SD卡)	Intent.ACTION_MEDIA_BAD_REMOVAL
插入外部储存装置（如SD卡）	Intent.ACTION_MEDIA_CHECKING
成功安装APK	Intent.ACTION_PACKAGE_ADDED
成功删除APK	Intent.ACTION_PACKAGE_REMOVED
重启设备	Intent.ACTION_REBOOT
屏幕被关闭	Intent.ACTION_SCREEN_OFF
屏幕被打开	Intent.ACTION_SCREEN_ON
关闭系统时	Intent.ACTION_SHUTDOWN
重启设备	Intent.ACTION_REBOOT

注：当使用系统广播时，只需要在注册广播接收者时定义相关的action即可，并不需要手动发送广播，当系统有相关操作时会自动进行系统广播

③. 有序广播（**Ordered Broadcast**）

- 定义 发送出去的广播被广播接收者按照先后顺序接收

> 有序是针对广播接收者而言的

- 广播接受者接收广播的顺序规则（同时面向静态和动态注册的广播接受者）

  1. 按照Priority属性值从大-小排序；
  2. Priority属性相同者，动态注册的广播优先；
- 特点

  1. 接收广播按顺序接收
  2. 先接收的广播接收者可以对广播进行截断，即后接收的广播接收者不再接收到此广播；
  3. 先接收的广播接收者可以对广播进行修改，那么后接收的广播接收者将接收到被修改后的广播
- 具体使用 有序广播的使用过程与普通广播非常类似，差异仅在于广播的发送方式：

```
sendOrderedBroadcast(intent);
```

④. App应用内广播（**Local Broadcast**）

- 背景 Android中的广播可以跨App直接通信（exported对于有intent-filter情况下默认值为true）

- 冲突 可能出现的问题：

  - 其他App针对性发出与当前App intent-filter相匹配的广播，由此导致当前App不断接收广播并处理；
  - 其他App注册与当前App一致的intent-filter用于接收广播，获取广播具体信息；即会出现安全性 & 效率性的问题。
- 解决方案 使用App应用内广播（Local Broadcast）

> 1. App应用内广播可理解为一种局部广播，广播的发送者和接收者都同属于一个App。
> 2. 相比于全局广播（普通广播），App应用内广播优势体现在：安全性高 & 效率高

- 具体使用1 - 将全局广播设置成局部广播

  1. 注册广播时将exported属性设置为*false*，使得非本App内部发出的此广播不被接收；

2. 在广播发送和接收时，增设相应权限permission，用于权限验证；
3. 发送广播时指定该广播接收器所在的包名，此广播将只会发送到此包中的App内与之相匹配的有效广播接收器中。

    > 通过 **intent.setPackage(packageName)** 指定报名

- 具体使用2 - 使用封装好的LocalBroadcastManager类 使用方式上与全局广播几乎相同，只是注册/取消注册广播接收器和发送广播时将参数的context变成了LocalBroadcastManager的单一实例

    > 注：对于LocalBroadcastManager方式发送的应用内广播，只能通过LocalBroadcastManager动态注册，不能静态注册

```
//注册应用内广播接收器
//步骤1：实例化BroadcastReceiver子类 & IntentFilter mBroadcastReceiver
mBroadcastReceiver = new mBroadcastReceiver();
IntentFilter intentFilter = new IntentFilter();

//步骤2：实例化LocalBroadcastManager的实例
localBroadcastManager = LocalBroadcastManager.getInstance(this);

//步骤3：设置接收广播的类型
intentFilter.addAction(android.net.conn.CONNECTIVITY_CHANGE);

//步骤4：调用LocalBroadcastManager单一实例的registerReceiver（）方法进行动态注册
localBroadcastManager.registerReceiver(mBroadcastReceiver, intentFilter);

//取消注册应用内广播接收器
localBroadcastManager.unregisterReceiver(mBroadcastReceiver);

//发送应用内广播
Intent intent = new Intent();
intent.setAction(BROADCAST_ACTION);
localBroadcastManager.sendBroadcast(intent);
```

⑤. 粘性广播（Sticky Broadcast）

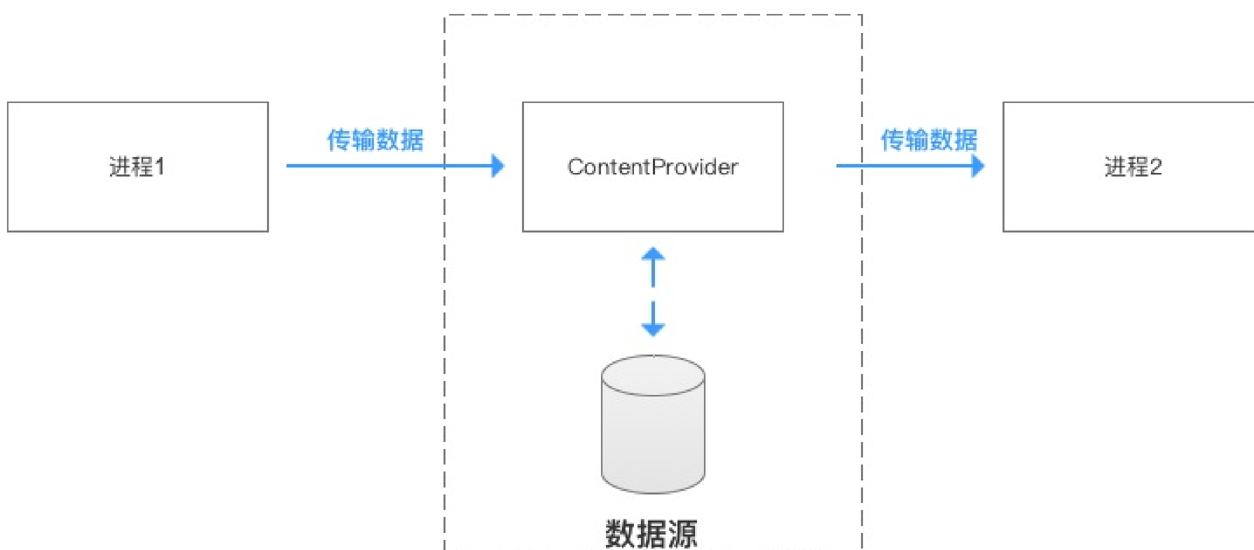
由于在Android5.0 & API 21中已经失效，所以不建议使用，在这里也不作过多的总结。

一、定义

ContentProvider，即内容提供者属于Android的四大组件之一。

二、作用

进程间 进行数据交互 & 共享，即跨进程通信



注：
1. ContentProvider = 中间者角色（搬运工），真正 存储&操作数据的数据源还是原来存储数据的方式（数据库、文件、xml或网络）
2. 数据源可以：数据库（如Sqlite）、文件、XML、网络等等

三、原理

`ContentProvider` 的底层是采用 `Android` 中的 `Binder` 机制

四、具体使用

关于 `ContentProvider` 的使用主要介绍以下内容：



4.1 统一资源标识符（URI）

- 定义：`Uniform Resource Identifier`，即统一资源标识符
- 作用：唯一标识 ContentProvider & 其中的数据

> 外界进程通过 URI 找到对应的ContentProvider & 其中的数据，再进行数据操作

- 具体使用

  URI分为 系统预置 & 自定义，分别对应系统内置的数据（如通讯录、日程表等等）和自定义数据库

  > 1. 关于 系统预置 `URI` 此处不作过多讲解，需要的同学可自行查看
  > 2. 此处主要讲解 自定义 `URI`

自定义URI = content:// com.carson.provider / User / 1

主题名 授权信息 表名 记录

- 主题（Schema）：Content Provider的URI前缀（Android 规定）
- 授权信息（Authority）：Content Provider的唯一标识符
- 表名（Path）：Content Provider 指向数据库中的某个表名
- 记录（ID）：表中的某个记录（若无指定，则返回全部记录）

```
// 设置URI
Uri uri = Uri.parse("content://com.carson.provider/User/1")
// 上述URI指向的资源是：名为 `com.carson.provider`的`ContentProvider
` 中表名 为`User` 中的 `id`为1的数据

// 特别注意：URI模式存在匹配通配符* & #

// *：匹配任意长度的任何有效字符的字符串
// 以下的URI 表示 匹配provider的任何内容
content://com.example.app.provider/*
// #：匹配任意长度的数字字符的字符串
// 以下的URI 表示 匹配provider中的table表的所有行
content://com.example.app.provider/table/#
```

4.2 MIME数据类型

- 解释：MIME：全称Multipurpose Internet Mail Extensions，多功能Internet 邮件扩充服务。它是一种多用途网际邮件扩充协议，在1992年最早应用于电子邮件系统，但后来也应用到浏览器。MIME类型就是设定某种扩展名的文件用一种应用程序来打开的方式类型，当该扩展名文件被访问的时候，浏览器会自动使用指定应用程序来打开。多用于指定一些客户端自定义的文件名，以及一些媒体文件打开方式。
- 作用：指定某个扩展名的文件用某种应用程序来打开 如指定 `.html` 文件采用 `text` 应用程序打开、指定 `.pdf` 文件采用 `flash` 应用程序打开
- 具体使用：

4.2.1 ContentProvider根据 URI 返回MIME类型

```
ContentProvider.geType(uri) ；
```

4.2.2 MIME类型组成 每种 `MIME` 类型 由2部分组成 = 类型 + 子类型

> MIME类型是 一个 包含2部分的字符串

```
text / html
// 类型 = text、子类型 = html

text/css
text/xml
application/pdf
```

4.2.3 MIME类型形式 `MIME` 类型有2种形式：

```
// 形式1：单条记录
vnd.android.cursor.item/自定义
// 形式2：多条记录（集合）
vnd.android.cursor.dir/自定义

// 注：
  // 1. vnd：表示父类型和子类型具有非标准的、特定的形式。
  // 2. 父类型已固定好（即不能更改），只能区别是单条还是多条记录
  // 3. 子类型可自定义
```

实例说明

```
<-- 单条记录 -->
  // 单个记录的MIME类型
  vnd.android.cursor.item/vnd.yourcompanyname.contenttype

  // 若一个Uri如下
  content://com.example.transportationprovider/trains/122
  // 则ContentProvider会通过ContentProvider.geType(url)返回以下MIME
类型
  vnd.android.cursor.item/vnd.example.rail


<-- 多条记录 -->
  // 多个记录的MIME类型
  vnd.android.cursor.dir/vnd.yourcompanyname.contenttype
  // 若一个Uri如下
  content://com.example.transportationprovider/trains
  // 则ContentProvider会通过ContentProvider.geType(url)返回以下MIME
类型
  vnd.android.cursor.dir/vnd.example.rail
```

4.3 ContentProvider类

4.3.1 组织数据方式

- ContentProvider主要以表格的形式组织数据

> 同时也支持文件数据，只是表格形式用得比较多

- 每个表格中包含多张表，每张表包含行 & 列，分别对应记录 & 字段

> 同数据库

4.3.2 主要方法

- 进程间共享数据的本质是：添加、删除、获取 & 修改（更新）数据
- 所以 `ContentProvider` 的核心方法也主要是上述4个作用

```
<-- 4个核心方法 -->
  public Uri insert(Uri uri, ContentValues values)
  // 外部进程向 ContentProvider 中添加数据

  public int delete(Uri uri, String selection, String[] selectio
nArgs)
  // 外部进程 删除 ContentProvider 中的数据

  public int update(Uri uri, ContentValues values, String select
ion, String[] selectionArgs)
  // 外部进程更新 ContentProvider 中的数据

  public Cursor query(Uri uri, String[] projection, String selec
tion, String[] selectionArgs,  String sortOrder)
  // 外部应用 获取 ContentProvider 中的数据

// 注：
  // 1. 上述4个方法由外部进程回调，并运行在ContentProvider进程的Binder
线程池中（不是主线程）
 // 2. 存在多线程并发访问，需要实现线程同步
   // a. 若ContentProvider的数据存储方式是使用SQLite & 一个，则不需要
，因为SQLite内部实现好了线程同步，若是多个SQLite则需要，因为SQL对象之间无
法进行线程同步
  // b. 若ContentProvider的数据存储方式是内存，则需要自己实现线程同步

<-- 2个其他方法 -->
public boolean onCreate()
// ContentProvider创建后 或 打开系统后其它进程第一次访问该ContentProvi
der时 由系统进行调用
// 注：运行在ContentProvider进程的主线程，故不能做耗时操作

public String getType(Uri uri)
// 得到数据类型，即返回当前 Url 所代表数据的MIME类型
```

- `Android` 为常见的数据（如通讯录、日程表等）提供了内置了默认的 `ContentProvider`
- 但也可根据需求自定义ContentProvider，但上述6个方法必须重写

> 本文主要讲解自定义`ContentProvider`

- `ContentProvider`类并不会直接与外部进程交互，而是通过`ContentResolver`类

4.4 ContentResolver类

4.41 作用

统一管理不同`ContentProvider`间的操作

> 1. 通过`URI`即可操作 不同的`ContentProvider`中的数据
> 2. 外部进程通过`ContentResolver`类 从而与`ContentProvider`类进行交互

4.2 为什么要使用通过**`ContentResolver`**类从而与**`ContentProvider`**类进行交互，而不直接访问**`ContentProvider`**类？

- 一般来说，一款应用要使用多个`ContentProvider`，若需要了解每个`ContentProvider`的不同实现从而再完成数据交互，操作成本高 **&** 难度大
- 所以再`ContentProvider`类上加多了一个`ContentResolver`类对所有的`ContentProvider`进行统一管理。

4.3 具体使用

`ContentResolver`类提供了与`ContentProvider`类相同名字 & 作用的4个方法

```
// 外部进程向 ContentProvider 中添加数据
public Uri insert(Uri uri, ContentValues values)

// 外部进程 删除 ContentProvider 中的数据
public int delete(Uri uri, String selection, String[] selectionA
rgs)

// 外部进程更新 ContentProvider 中的数据
public int update(Uri uri, ContentValues values, String selectio
n, String[] selectionArgs)

// 外部应用 获取 ContentProvider 中的数据
public Cursor query(Uri uri, String[] projection, String selecti
on, String[] selectionArgs, String sortOrder)
```

- 实例说明

```
// 使用ContentResolver前，需要先获取ContentResolver
// 可通过在所有继承Context的类中 通过调用getContentResolver()来获得Con
tentResolver
ContentResolver resolver =  getContentResolver();

// 设置ContentProvider的URI
Uri uri = Uri.parse("content://cn.scu.myprovider/user");

// 根据URI 操作 ContentProvider中的数据
// 此处是获取ContentProvider中 user表的所有记录
Cursor cursor = resolver.query(uri, null, null, null, "userid de
sc");
```

`Android` 提供了3个用于辅助 `ContentProvide` 的工具类：

- `ContentUris`
- `UriMatcher`
- `ContentObserver`

4.5 ContentUris类

- 作用：操作 `URI`
- 具体使用 核心方法有两个：`withAppendedId（）` & `parseId（）`

```
// withAppendedId（）作用：向URI追加一个id
Uri uri = Uri.parse("content://cn.scu.myprovider/user")
Uri resultUri = ContentUris.withAppendedId(uri, 7);
// 最终生成后的Uri为：content://cn.scu.myprovider/user/7

// parseId（）作用：从URL中获取ID
Uri uri = Uri.parse("content://cn.scu.myprovider/user/7")
long personid = ContentUris.parseId(uri);
//获取的结果为:7
```

4.6 UriMatcher类

- 作用
  1. 在 `ContentProvider` 中注册 `URI`
  2. 根据 `URI` 匹配 `ContentProvider` 中对应的数据表
- 具体使用

```
// 步骤1：初始化UriMatcher对象
UriMatcher matcher = new UriMatcher(UriMatcher.NO_MATCH);
//常量UriMatcher.NO_MATCH  = 不匹配任何路径的返回码
// 即初始化时不匹配任何东西

// 步骤2：在ContentProvider 中注册URI（addURI（））
int URI_CODE_a = 1；
int URI_CODE_b = 2；
matcher.addURI("cn.scu.myprovider", "user1", URI_CODE_a);
matcher.addURI("cn.scu.myprovider", "user2", URI_CODE_b);
// 若URI资源路径 = content://cn.scu.myprovider/user1 ，则返回注册码URI_CODE_a
// 若URI资源路径 = content://cn.scu.myprovider/user2 ，则返回注册码URI_CODE_b

// 步骤3：根据URI 匹配 URI_CODE，从而匹配ContentProvider中相应的资源（match（））

@Override
public String getType (Uri uri){
    Uri uri = Uri.parse(" content://cn.scu.myprovider/user1");

    switch (matcher.match(uri)) {
        // 根据URI匹配的返回码是URI_CODE_a
        // 即matcher.match(uri) == URI_CODE_a
        case URI_CODE_a:
            return tableNameUser1;
        // 如果根据URI匹配的返回码是URI_CODE_a，则返回ContentProvider中的名为tableNameUser1的表
        case URI_CODE_b:
            return tableNameUser2;
        // 如果根据URI匹配的返回码是URI_CODE_b，则返回ContentProvider中的名为tableNameUser2的表
    }
}
```

4.7 ContentObserver类

- 定义：内容观察者
- 作用：观察 Uri引起ContentProvider 中的数据变化 & 通知外界（即访问该数据访问者）

> 当 `ContentProvider` 中的数据发生变化（增、删 & 改）时，就会触发该 `ContentObserver` 类

- 具体使用

```
// 步骤1：注册内容观察者ContentObserver
getContentResolver().registerContentObserver（uri）；
// 通过ContentResolver类进行注册，并指定需要观察的URI

// 步骤2：当该URI的ContentProvider数据发生变化时，通知外界（即访问该ContentProvider数据的访问者）
public class UserContentProvider extends ContentProvider {
    public Uri insert(Uri uri, ContentValues values) {
        db.insert("user", "userid", values);
        getContext().getContentResolver().notifyChange(uri, null);
        // 通知访问者
    }
}

// 步骤3：解除观察者
getContentResolver().unregisterContentObserver（uri）；
// 同样需要通过ContentResolver类进行解除
```

至此，关于 `ContentProvider` 的使用已经讲解完毕

五、 实例说明

- 由于 `ContentProvider` 不仅常用于进程间通信，同时也适用于进程内通信
- 所以本实例会采用ContentProvider讲解：
    1. 进程内通信
    2. 进程间通信

- 实例说明：采用的数据源是 `Android` 中的 `SQLite` 数据库

5.1 进程内通信

- 步骤说明：
    1. 创建数据库类
    2. 自定义 `ContentProvider` 类
    3. 注册 创建的 `ContentProvider` 类
    4. 进程内访问 `ContentProvider` 的数据
- 具体使用

步骤**1**：创建数据库类 **DBHelper.java**

```
public class DBHelper extends SQLiteOpenHelper {

    // 数据库名
    private static final String DATABASE_NAME = "finch.db";

    // 表名
    public static final String USER_TABLE_NAME = "user";
    public static final String JOB_TABLE_NAME = "job";

    private static final int DATABASE_VERSION = 1;
    //数据库版本号

    public DBHelper(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {

        // 创建两个表格:用户表 和职业表
        db.execSQL("CREATE TABLE IF NOT EXISTS " + USER_TABLE_NA
ME + "(_id INTEGER PRIMARY KEY AUTOINCREMENT," + " name TEXT)");
        db.execSQL("CREATE TABLE IF NOT EXISTS " + JOB_TABLE_NAM
E + "(_id INTEGER PRIMARY KEY AUTOINCREMENT," + " job TEXT)");
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int
 newVersion)   {

    }
}
```

步骤2：自定义 **ContentProvider** 类

```
public class MyProvider extends ContentProvider {

    private Context mContext;
    DBHelper mDbHelper = null;
```

```
    SQLiteDatabase db = null;

    public static final String AUTOHORITY = "cn.scu.myprovider";
    // 设置ContentProvider的唯一标识

    public static final int User_Code = 1;
    public static final int Job_Code = 2;

    // UriMatcher类使用:在ContentProvider 中注册URI
    private static final UriMatcher mMatcher;

    static {
        mMatcher = new UriMatcher(UriMatcher.NO_MATCH);
        // 初始化
        mMatcher.addURI(AUTOHORITY, "user", User_Code);
        mMatcher.addURI(AUTOHORITY, "job", Job_Code);
        // 若URI资源路径 = content://cn.scu.myprovider/user ，则返
回注册码User_Code
        // 若URI资源路径 = content://cn.scu.myprovider/job ，则返回
注册码Job_Code
    }

    // 以下是ContentProvider的6个方法

    /**
     * 初始化ContentProvider
     */
    @Override
    public boolean onCreate() {

        mContext = getContext();
        // 在ContentProvider创建时对数据库进行初始化
        // 运行在主线程，故不能做耗时操作,此处仅作展示
        mDbHelper = new DBHelper(getContext());
        db = mDbHelper.getWritableDatabase();

        // 初始化两个表的数据(先清空两个表,再各加入一个记录)
        db.execSQL("delete from user");
        db.execSQL("insert into user values(1,'Carson');");
        db.execSQL("insert into user values(2,'Kobe');");
```

```
        db.execSQL("delete from job");
        db.execSQL("insert into job values(1,'Android');");
        db.execSQL("insert into job values(2,'iOS');");

        return true;
    }

    /**
     * 添加数据
     */
    @Override
    public Uri insert(Uri uri, ContentValues values) {

        // 根据URI匹配 URI_CODE，从而匹配ContentProvider中相应的表名
        // 该方法在最下面
        String table = getTableName(uri);

        // 向该表添加数据
        db.insert(table, null, values);

        // 当该URI的ContentProvider数据发生变化时，通知外界（即访问该Co
ntentProvider数据的访问者）
        mContext.getContentResolver().notifyChange(uri, null);

//          // 通过ContentUris类从URL中获取ID
//          long personid = ContentUris.parseId(uri);
//          System.out.println(personid);

        return uri;
    }

    /**
     * 查询数据
     */
    @Override
    public Cursor query(Uri uri, String[] projection, String sel
ection,
                        String[] selectionArgs, String sortOrder)
 {
```

```
        // 根据URI匹配 URI_CODE，从而匹配ContentProvider中相应的表名
        // 该方法在最下面
        String table = getTableName(uri);

//          // 通过ContentUris类从URL中获取ID
//          long personid = ContentUris.parseId(uri);
//          System.out.println(personid);

        // 查询数据
        return db.query(table, projection, selection, selectionA
rgs, null, null, sortOrder, null);
    }

    /**
     * 更新数据
     */
    @Override
    public int update(Uri uri, ContentValues values, String sele
ction,
                      String[] selectionArgs) {
        // 由于不展示,此处不作展开
        return 0;
    }

    /**
     * 删除数据
     */
    @Override
    public int delete(Uri uri, String selection, String[] select
ionArgs) {
        // 由于不展示,此处不作展开
        return 0;
    }

    @Override
    public String getType(Uri uri) {

        // 由于不展示,此处不作展开
        return null;
    }
```

```
    /**
     * 根据URI匹配 URI_CODE，从而匹配ContentProvider中相应的表名
     */
    private String getTableName(Uri uri) {
        String tableName = null;
        switch (mMatcher.match(uri)) {
            case User_Code:
                tableName = DBHelper.USER_TABLE_NAME;
                break;
            case Job_Code:
                tableName = DBHelper.JOB_TABLE_NAME;
                break;
        }
        return tableName;
    }
}
```

步骤3：注册 创建的 **ContentProvider**类 **AndroidManifest.xml**

```
<provider android:name="MyProvider"
  android:authorities="cn.scu.myprovider"/>
```

步骤4：进程内访问 **ContentProvider**中的数据

MainActivity.java

```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        /**
         * 对user表进行操作
         */
```

```
        // 设置URI
        Uri uri_user = Uri.parse("content://cn.scu.myprovider/us
er");

        // 插入表中数据
        ContentValues values = new ContentValues();
        values.put("_id", 3);
        values.put("name", "Iverson");


        // 获取ContentResolver
        ContentResolver resolver =  getContentResolver();
        // 通过ContentResolver 根据URI 向ContentProvider中插入数据
        resolver.insert(uri_user,values);

        // 通过ContentResolver 向ContentProvider中查询数据
        Cursor cursor = resolver.query(uri_user, new String[]{"_
id","name"}, null, null, null);
        while (cursor.moveToNext()){
            System.out.println("query book:" + cursor.getInt(0) +
" "+ cursor.getString(1));
            // 将表中数据全部输出
        }
        cursor.close();
        // 关闭游标

        /**
         * 对job表进行操作
         */
        // 和上述类似,只是URI需要更改,从而匹配不同的URI CODE,从而找到不
同的数据资源
        Uri uri_job = Uri.parse("content://cn.scu.myprovider/job"
);

        // 插入表中数据
        ContentValues values2 = new ContentValues();
        values2.put("_id", 3);
        values2.put("job", "NBA Player");

        // 获取ContentResolver
```

```
        ContentResolver resolver2 =  getContentResolver();
        // 通过ContentResolver 根据URI 向ContentProvider中插入数据
        resolver2.insert(uri_job,values2);

        // 通过ContentResolver 向ContentProvider中查询数据
        Cursor cursor2 = resolver2.query(uri_job, new String[]{"_id","job"}, null, null, null);
        while (cursor2.moveToNext()){
            System.out.println("query job:" + cursor2.getInt(0) + " "+ cursor2.getString(1));
            // 将表中数据全部输出
        }
        cursor2.close();
        // 关闭游标
}
}
```

结果

```
06-06 01:30:54.681 5746-5746/scut.carson_ho.contentprovider I/System.out: query book:1 Carson
06-06 01:30:54.681 5746-5746/scut.carson_ho.contentprovider I/System.out: query book:2 Kobe
06-06 01:30:54.681 5746-5746/scut.carson_ho.contentprovider I/System.out: query book:3 Iverson
06-06 01:30:54.705 5746-5746/scut.carson_ho.contentprovider I/System.out: query job:1 Android
06-06 01:30:54.705 5746-5746/scut.carson_ho.contentprovider I/System.out: query job:2 iOS
06-06 01:30:54.705 5746-5746/scut.carson_ho.contentprovider I/System.out: query job:3 NBA Player
```

5.2 进程间进行数据共享

- 实例说明：本文需要创建2个进程，即创建两个工程，作用如下



进程1

使用步骤如下：

1. 创建数据库类
2. 自定义 `ContentProvider` 类
3. 注册 创建的 `ContentProvider` 类

前2个步骤同上例相同，此处不作过多描述，此处主要讲解步骤3.

步骤3：注册 创建的 **ContentProvider**类 *AndroidManifest.xml*

```
<provider
                android:name="MyProvider"
                android:authorities="scut.carson_ho.myprovider"

                // 声明外界进程可访问该Provider的权限（读 & 写）
                android:permission="scut.carson_ho.PROVIDER"


                // 权限可细分为读 & 写的权限
                // 外界需要声明同样的读 & 写的权限才可进行相应操作，否则会
报错
                // android:readPermisson = "scut.carson_ho.Read"
                // android:writePermisson = "scut.carson_ho.Write
"

                // 设置此provider是否可以被其他进程使用
                android:exported="true"

  />

// 声明本应用 可允许通信的权限
    <permission android:name="scut.carson_ho.Read" android:prote
ctionLevel="normal"/>
    // 细分读 & 写权限如下，但本Demo直接采用全权限
    // <permission android:name="scut.carson_ho.Write" android:p
rotectionLevel="normal"/>
    // <permission android:name="scut.carson_ho.PROVIDER" androi
d:protectionLevel="normal"/>
```

至此，进程1创建完毕，即创建 `ContentProvider` & 数据 准备好了。

进程2

步骤1：声明可访问的权限

AndroidManifest.xml

```
    // 声明本应用可允许通信的权限（全权限）
    <uses-permission android:name="scut.carson_ho.PROVIDER"/>

    // 细分读 & 写权限如下，但本Demo直接采用全权限
    // <uses-permission android:name="scut.carson_ho.Read"/>
    //  <uses-permission android:name="scut.carson_ho.Write"/>

// 注：声明的权限必须与进程1中设置的权限对应
```

步骤2：访问 **ContentProvider**的类

```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        /**
         * 对user表进行操作
         */

        // 设置URI
        Uri uri_user = Uri.parse("content://scut.carson_ho.mypro
vider/user");

        // 插入表中数据
        ContentValues values = new ContentValues();
        values.put("_id", 4);
        values.put("name", "Jordan");


        // 获取ContentResolver
        ContentResolver resolver =  getContentResolver();
        // 通过ContentResolver 根据URI 向ContentProvider中插入数据
        resolver.insert(uri_user,values);

        // 通过ContentResolver 向ContentProvider中查询数据
        Cursor cursor = resolver.query(uri_user, new String[]{"_
```

```
id","name"}, null, null, null);
        while (cursor.moveToNext()){
            System.out.println("query book:" + cursor.getInt(0) +
" "+ cursor.getString(1));
            // 将表中数据全部输出
        }
        cursor.close();
        // 关闭游标

        /**
         * 对job表进行操作
         */
        // 和上述类似,只是URI需要更改,从而匹配不同的URI CODE,从而找到不
同的数据资源
        Uri uri_job = Uri.parse("content://scut.carson_ho.myprov
ider/job");

        // 插入表中数据
        ContentValues values2 = new ContentValues();
        values2.put("_id", 4);
        values2.put("job", "NBA Player");

        // 获取ContentResolver
        ContentResolver resolver2 =  getContentResolver();
        // 通过ContentResolver 根据URI 向ContentProvider中插入数据
        resolver2.insert(uri_job,values2);

        // 通过ContentResolver 向ContentProvider中查询数据
        Cursor cursor2 = resolver2.query(uri_job, new String[]{"
_id","job"}, null, null, null);
        while (cursor2.moveToNext()){
            System.out.println("query job:" + cursor2.getInt(0) +
" "+ cursor2.getString(1));
            // 将表中数据全部输出
        }
        cursor2.close();
        // 关闭游标
    }
}
```

结果展示

在进程展示时，需要先运行准备数据的进程**1**，再运行需要访问数据的进程**2**

1. 运行准备数据的进程1 在进程1中，我们准备好了一系列数据

```
06-06 03:35:47.792 24121-24121/scut.carson_ho.contentprovider I/System.out: query book:1 Carson
06-06 03:35:47.792 24121-24121/scut.carson_ho.contentprovider I/System.out: query book:2 Kobe
06-06 03:35:47.792 24121-24121/scut.carson_ho.contentprovider I/System.out: query book:3 Iverson
06-06 03:35:47.872 24121-24121/scut.carson_ho.contentprovider I/System.out: query job:1 Android
06-06 03:35:47.872 24121-24121/scut.carson_ho.contentprovider I/System.out: query job:2 iOS
06-06 03:35:47.872 24121-24121/scut.carson_ho.contentprovider I/System.out: query job:3 NBA Player
```

2. 运行需要访问数据的进程2 在进程2中，我们先向 `ContentProvider` 中插入数据，再查询数据

```
06-06 03:37:02.488 26442-26442/scut.carson_ho.contentprovider2 I/System.out: query book:1 Carson
06-06 03:37:02.488 26442-26442/scut.carson_ho.contentprovider2 I/System.out: query book:2 Kobe
06-06 03:37:02.488 26442-26442/scut.carson_ho.contentprovider2 I/System.out: query book:3 Iverson
06-06 03:37:02.488 26442-26442/scut.carson_ho.contentprovider2 I/System.out: query book:4 Jordan      在进程2中插入的数据
06-06 03:37:02.512 26442-26442/scut.carson_ho.contentprovider2 I/System.out: query job:1 Android
06-06 03:37:02.516 26442-26442/scut.carson_ho.contentprovider2 I/System.out: query job:2 iOS
06-06 03:37:02.516 26442-26442/scut.carson_ho.contentprovider2 I/System.out: query job:3 NBA Player
06-06 03:37:02.516 26442-26442/scut.carson_ho.contentprovider2 I/System.out: query job:4 NBA Player  在进程2中插入的数据
```

六、优点

6.1 安全

`ContentProvider` 为应用间的数据交互提供了一个安全的环境：允许把自己的应用数据根据需求开放给 其他应用 进行 增、删、改、查，而不用担心因为直接开放数据库权限而带来的安全问题

6.2 访问简单 & 高效

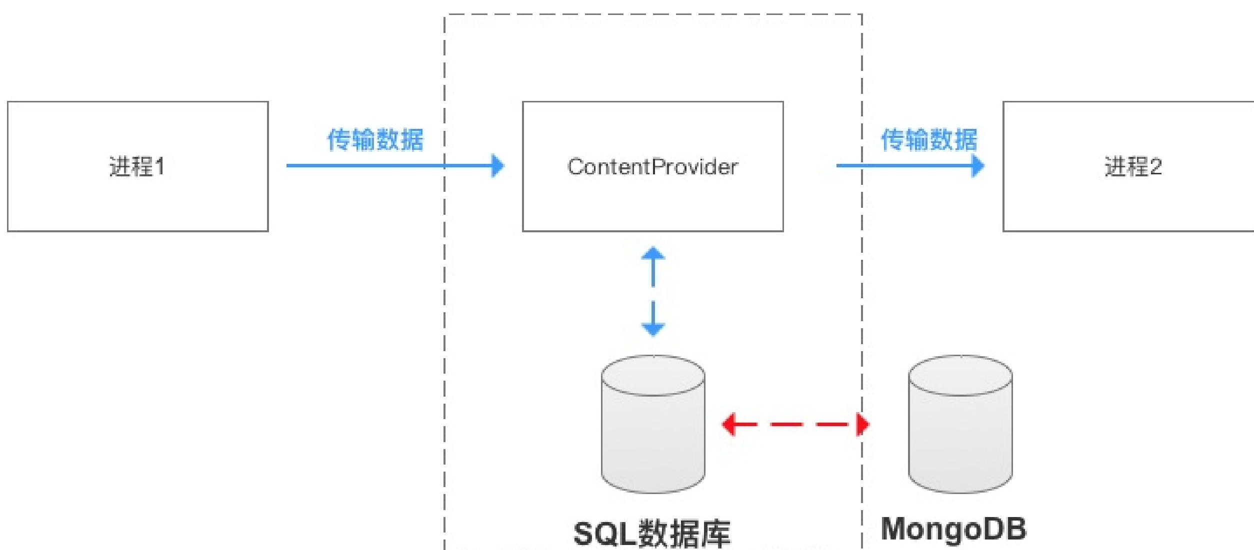
对比于其他对外共享数据的方式，数据访问方式会因数据存储的方式而不同：

- 采用 文件方式 对外共享数据，需要进行文件操作读写数据；
- 采用 `Sharedpreferences` 共享数据，需要使用sharedpreferences API读写数据

这使得访问数据变得复杂 & 难度大。

- 而采用ContentProvider方式，其解耦了底层数据的存储方式，使得无论底层数据存储采用何种方式，外界对数据的访问方式都是统一的，这使得访问简单 & 高效

> 如一开始数据存储方式 采用 `SQLite` 数据库，后来把数据库换成 `MongoDB` ，也不会对上层数据 `ContentProvider` 使用代码产生影响



七、 总结

- 我用一张图总结本文内容



一、目录

- 什么是Fragment
- Fragment的生命周期
- Fragment的使用方式
- 什么是Fragment的回退栈？【重要】
- Fragment与Activity之间的通信【难点】
- Fragment与Activity通信的优化【超难点】
- 如何处理运行时配置发生变化【以屏幕翻转为例】

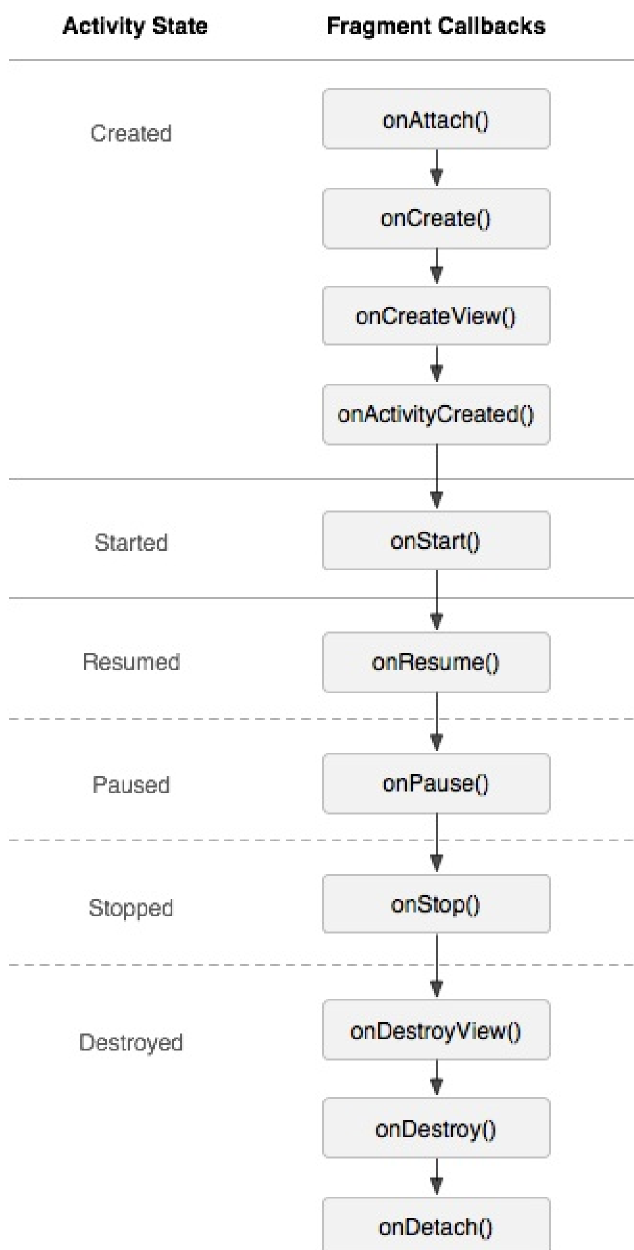
二、Fragment详解

1. 什么是Fragment ？

你可以简单的理解为，Fragment是显示在Activity中的Activity。它可以显示在Activity中，然后它也可以显示出一些内容。因为它拥有自己的生命周期，可以接受处理用户的事件，并且你可以在一个Activity中动态的添加，替换，移除不同的Fragment，因此对于信息的展示具有很大的便利性。

2. Fragment的生命周期

因为Fragment是依附于Activity存在的，因此它的生命周期收到Activity的生命周期影响



Fragment比Activity多了几个生命周期的回调方法

- onAttach(Activity) 当Fragment与Activity发生关联的时候调用
- onCreateView(LayoutInflater, ViewGroup, Bundle) 创建该Fragment的视图
- onActivityCreated(Bundle) 当Activity的onCreated方法返回时调用
- onDestroyView() 与onCreateView方法相对应，当该Fragment的视图被移除时调用
- onDetach() 与onAttach方法相对应，当Fragment与Activity取消关联时调用

PS：注意：除了onCreateView，其他的所有方法如果你重写了，必须调用父类对于该方法的实现

3. Fragment的使用方式

静态使用Fragment

步骤：

① 创建一个类继承Fragment，重写onCreateView方法，来确定Fragment要显示的布局

② 在Activity中声明该类，与普通的View对象一样

代码演示

MyFragment对应的布局文件item_fragment.xml

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/re
s/android"
    android:background="@color/colorAccent"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <ImageView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true"
        android:src="@mipmap/ic_launcher" />

</RelativeLayout>
```

继承Frgmanet的类MyFragment【请注意导包的时候导v4的Fragment的包】

```
public class MyFragment extends Fragment {
    public View onCreateView(LayoutInflater inflater, ViewGroup
container, Bundle savedInstanceState) {
        /*
        * 参数1：布局文件的id
        * 参数2：容器
        * 参数3：是否将这个生成的View添加到这个容器中去
        * 作用是将布局文件封装在一个View对象中，并填充到此Fragment中
        * */
        View v = inflater.inflate(R.layout.item_fragment, contai
ner, false);
        return v;
    }
}
```

Activity对应的布局文件

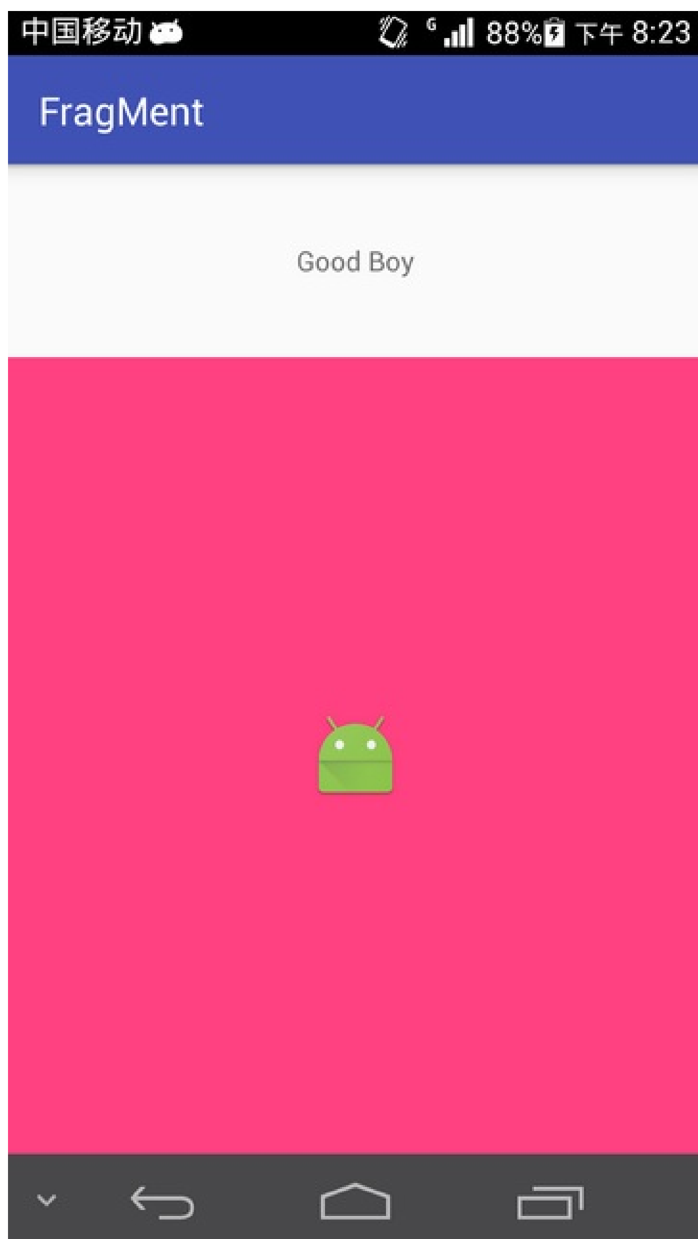
```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context="com.usher.fragment.MainActivity">

    <TextView
        android:layout_width="match_parent"
        android:layout_height="100dp"
        android:gravity="center"
        android:text="Good Boy" />

    <fragment
        android:id="@+id/myfragment"
        android:name="com.usher.fragment.MyFragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

</LinearLayout>
```

运行效果图



动态使用Fragment

实现点击不同的按钮，在Activity中显示不同的Fragment

代码演示

Fragment对应的布局文件item_fragment.xml

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/re
s/android"
    android:background="@color/colorAccent" //背景红色
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <ImageView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true"
        android:src="@mipmap/ic_launcher" />

</RelativeLayout>
```

继承Frgmanet的类MyFragment

```
public class MyFragment extends Fragment {
    public View onCreateView(LayoutInflater inflater, ViewGroup
container, Bundle savedInstanceState) {
        View v = inflater.inflate(R.layout.item_fragment, contai
ner, false);
        return v;
    }
}
```

Fragment2对应的布局文件item_fragment2.xml

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/re
s/android"
    android:background="@color/colorPrimary" //背景蓝色
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <ImageView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true"
        android:src="@mipmap/ic_launcher" />

</RelativeLayout>
```

继承Fragment2的类

```
public class MyFragment extends Fragment {
    public View onCreateView(LayoutInflater inflater, ViewGroup
container, Bundle savedInstanceState) {
        View v = inflater.inflate(R.layout.item_fragment2, conta
iner, false);
        return v;
    }
}
```

MainActivity对应的布局文件

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context="com.usher.fragment.MainActivity">

    <Button
        android:id="@+id/bt_red"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Red" />

    <Button
        android:id="@+id/bt_blue"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Blue" />

    <FrameLayout
        android:id="@+id/myframelayout"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

</LinearLayout>
```

MainActivity类

```
public class MainActivity extends AppCompatActivity {

    private Button bt_red;
    private Button bt_blue;
    private FragmentManager manager;
    private MyFragment fragment1;
    private MyFragment2 fragment2;
```

```
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        initView();

        fragment1 = new MyFragment();
        fragment2 = new MyFragment2();

        //初始化FragmentManager对象
        manager = getSupportFragmentManager();

        //使用FragmentManager对象用来开启一个Fragment事务
        FragmentTransaction transaction = manager.beginTransacti
on();

        //默认显示fragment1
        transaction.add(R.id.myframelayout, fragment1).commit();

        //对bt_red设置监听
        bt_red.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                FragmentTransaction transaction = manager.beginT
ransaction();
                transaction.replace(R.id.myframelayout, fragment
1).commit();
            }
        });

        //对bt_blue设置监听
        bt_blue.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                FragmentTransaction transaction = manager.beginT
ransaction();
                transaction.replace(R.id.myframelayout, fragment
2).commit();
            }
```

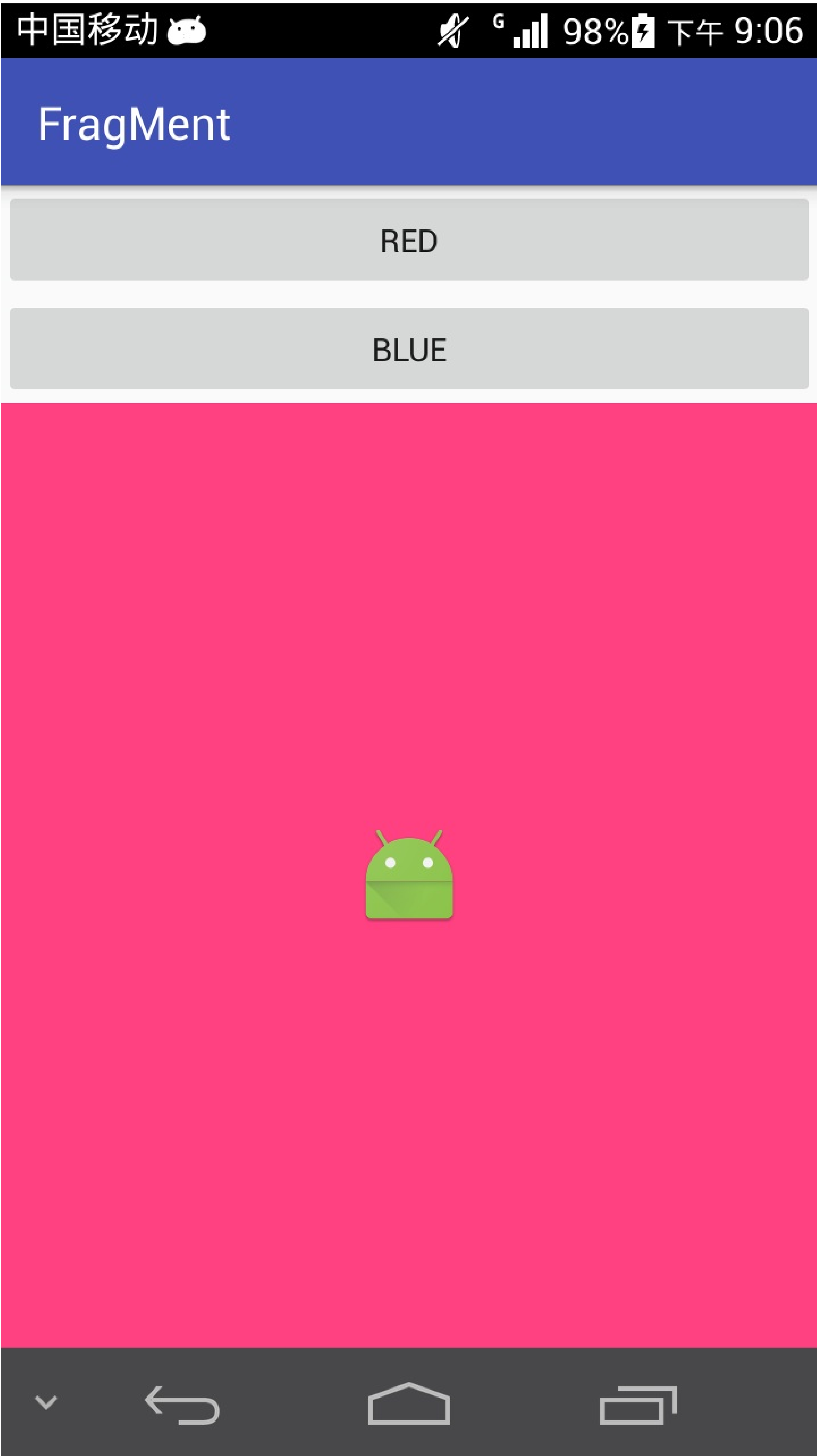
```
        });
    }

    private void initView() {
        bt_red = (Button) findViewById(R.id.bt_red);
        bt_blue = (Button) findViewById(R.id.bt_blue);
    }

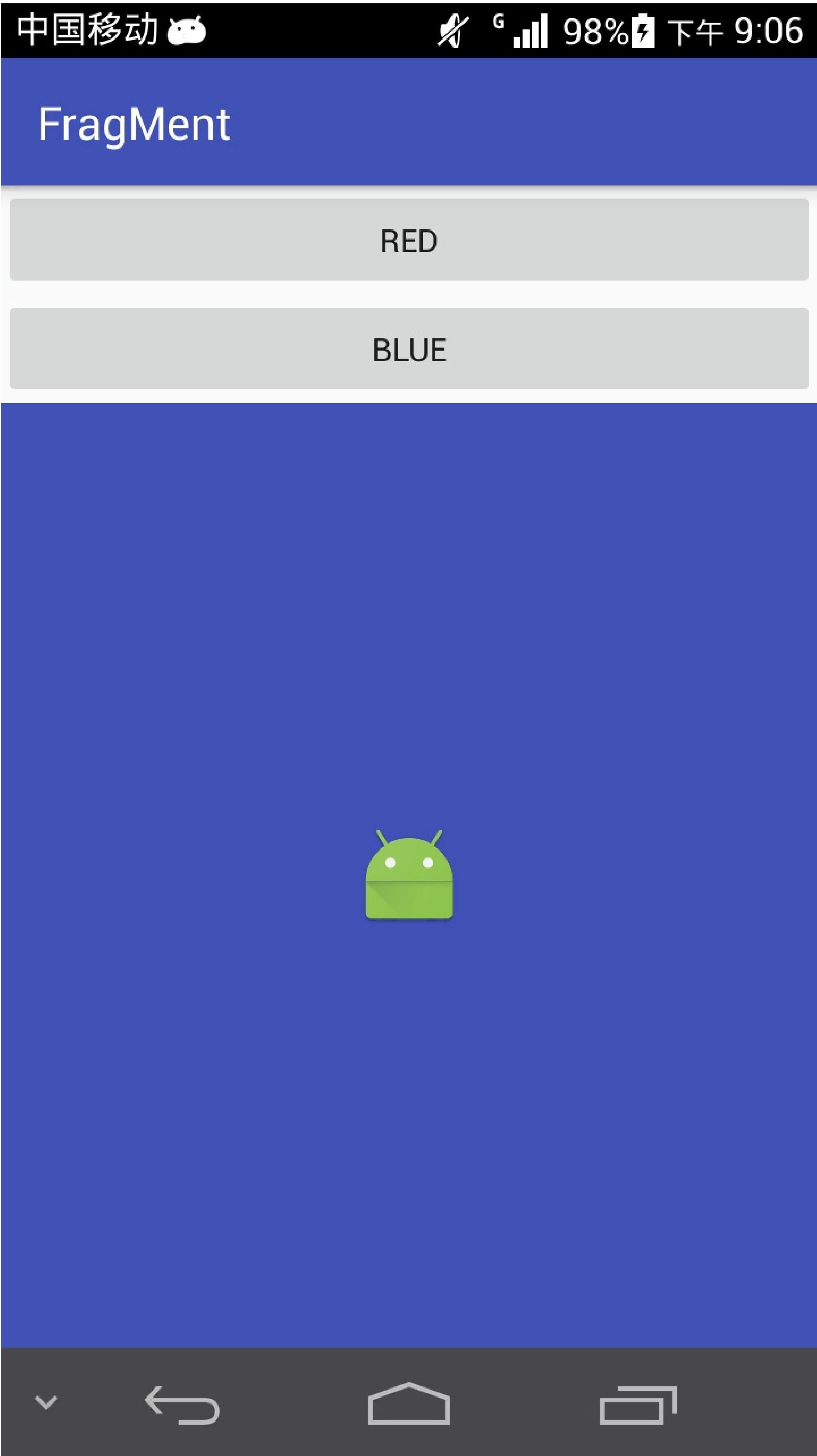
}
```

显示效果

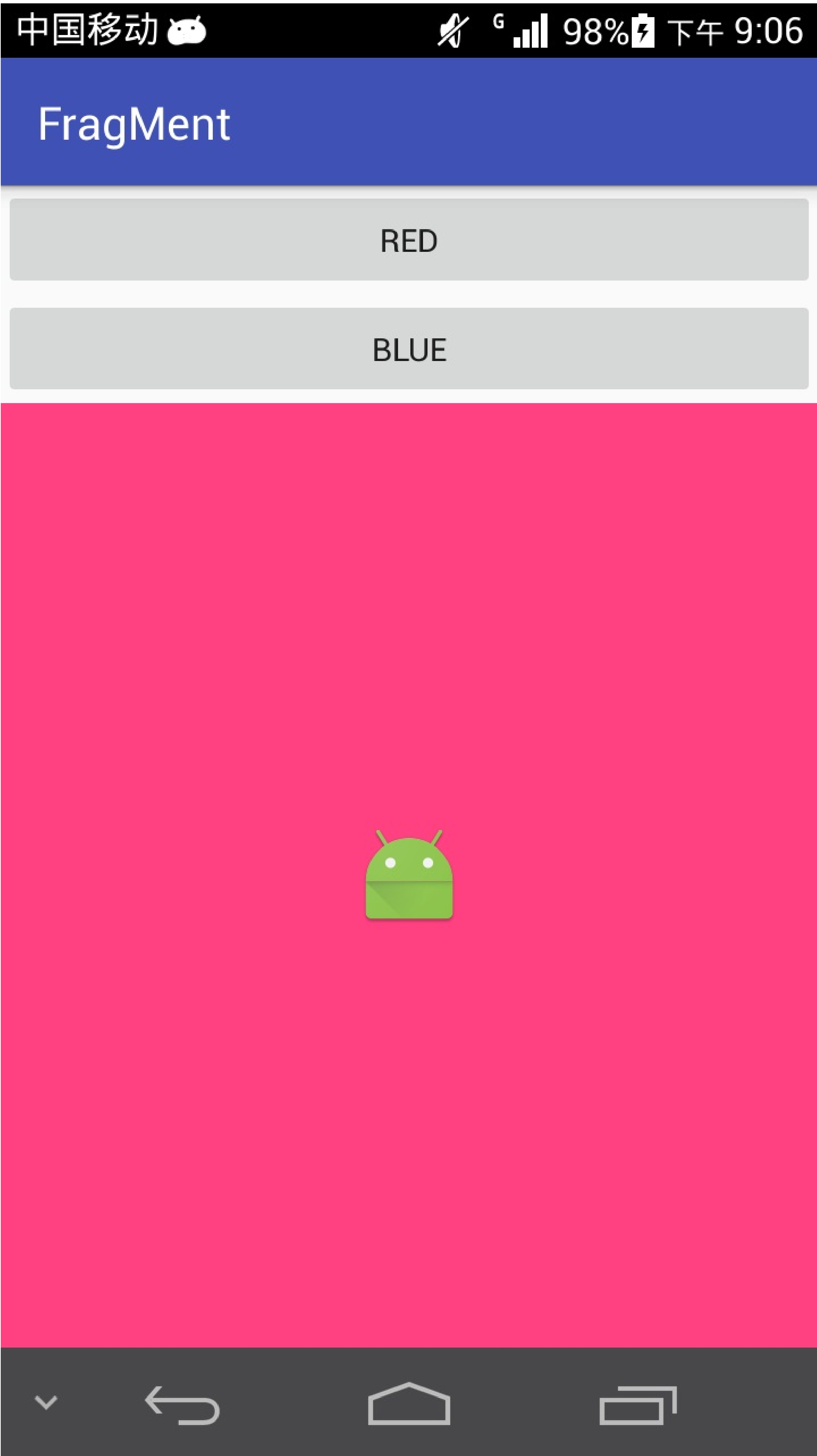
默认显示


中国移动
98%
下午 9:06
FragMent
RED
BLUE

点击BLUE按钮时


中国移动
98%
下午 9:06
FragMent
RED
BLUE

点击RED按钮时


中国移动
98%
下午 9:06
FragMent
RED
BLUE

以上代码我写的比较臃肿但是比较容易看明白：

① 在Acitivity对应的布局中写上一个FramLayout控件，此空间的作用是当作Fragment的容器，Fragment通过FrameLayout显示在Acitivity里，这两个单词容易混淆，请注意

② 准备好你的Fragment，然后再Activity中实例化，v4包的Fragment是通过getSupportFragmentManager()方法新建Fragment管理器对象，此处不讨论app包下的Fragment

③ 然后通过Fragment管理器对象调用beginTransaction()方法，实例化FragmentTransaction对象，有人称之为事务

④ FragmentTransaction对象【以下直接用transaction代替】，transaction的方法主要有以下几种：

- transaction.add() 向Activity中添加一个Fragment
- transaction.remove() 从Activity中移除一个Fragment，如果被移除的Fragment没有添加到回退栈（回退栈后面会详细说），这个Fragment实例将会被销毁
- transaction.replace() 使用另一个Fragment替换当前的，实际上就是remove()然后add()的合体
- transaction.hide() 隐藏当前的Fragment，仅仅是设为不可见，并不会销毁
- transaction.show() 显示之前隐藏的Fragment
- detach() 会将view从UI中移除,和remove()不同,此时fragment的状态依然由FragmentManager维护
- attach() 重建view视图，附加到UI上并显示
- ransatcion.commit() 提交事务

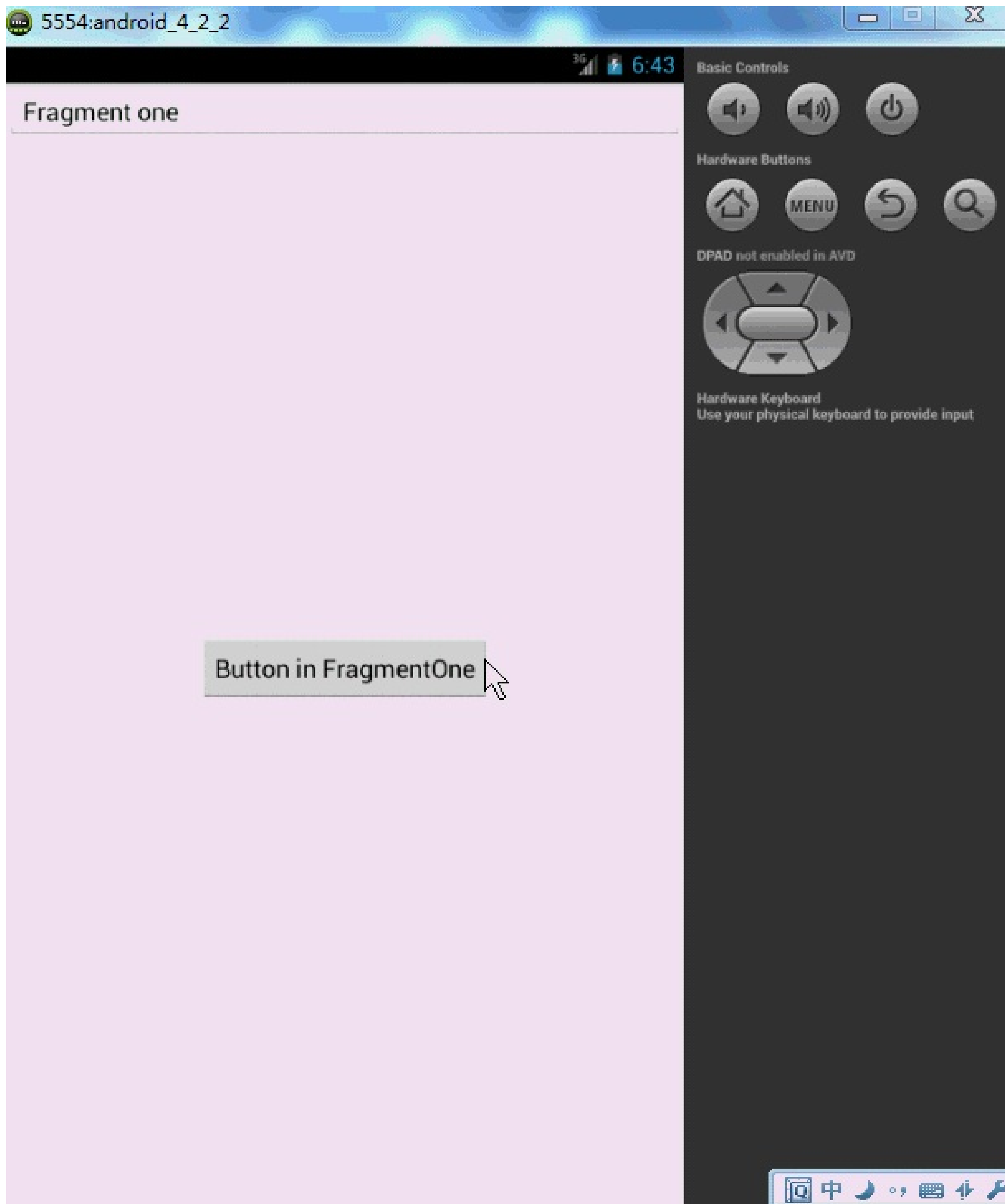
注意：在add/replace/hide/show以后都要commit其效果才会在屏幕上显示出来

4. 什么是Fragment的回退栈？

- Fragment的回退栈是用来保存每一次Fragment事务发生的变化 如果你将Fragment任务添加到回退栈，当用户点击后退按钮时，将看到上一次的保存的Fragment。一旦Fragment完全从后退栈中弹出，用户再次点击后退键，则退出当前Activity

  那么这句话要怎么理解？

首先来看一下这个东西：



- 首先显示第一个FragmentOne页面有一个Button in FragmentOne，上面有个输入框显示的是Fragment One
- 然后输入change，点击Button in FragmentOne，然后显示第二个Fragment，里面有一个Button in FragmentTwo，一个输入框显示Fragment Two
- 输入change，点击按钮，显示第三个Fragment，上面有个Button in FragmentThree，点击按钮显示出一个Toast
- 【注意】点击返回键，跳转到前一个FragmentTwo，这个时候可以看到上面的输入框中显示的是Fragment Two change，也就是说保留了我们离开这个

Fragment时候他所呈现的状态
- 【注意】再点击返回键，跳转到FragmentOne，但是这个时候我们可以看到上面的输入框中只有Fragment One，并没有change这几个字母

那么原因是什么？

这里先要学习一个方法：**FragmentTransaction.addToBackStack(String)**【把当前事务的变化情况添加到回退栈】

代码如下：

MainActivity的布局文件

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/re
s/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <FrameLayout
        android:id="@+id/id_content"
        android:layout_width="match_parent"
        android:layout_height="match_parent" >
    </FrameLayout>

</RelativeLayout>
```

MainActivity.java文件【这里添加的是app包下的Fragment，推荐v4包下的】

```
public class MainActivity extends Activity {

    protected void onCreate(Bundle savedInstanceState){
        super.onCreate(savedInstanceState);
        requestWindowFeature(Window.FEATURE_NO_TITLE);
        setContentView(R.layout.activity_main);

        FragmentManager fm = getFragmentManager();
        FragmentTransaction tx = fm.beginTransaction();
        tx.add(R.id.id_content, new FragmentOne(),"ONE");
        tx.commit();
    }

}
```

FragmentOne.class文件

```
public class FragmentOne extends Fragment implements OnClickList
ener {

    private Button mBtn;

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup
container,
            Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.fragment_one, cont
ainer, false);
        mBtn = (Button) view.findViewById(R.id.id_fragment_one_b
tn);
        mBtn.setOnClickListener(this);
        return view;
    }

    @Override
    public void onClick(View v) {
        FragmentTwo fTwo = new FragmentTwo();
        FragmentManager fm = getFragmentManager();
        FragmentTransaction tx = fm.beginTransaction();
        tx.replace(R.id.id_content, fTwo, "TWO");
        tx.addToBackStack(null);
        tx.commit();
    }

}
```

Fragment的点击事件里写的是replace方法，相当于remove和add的合体，并且如果不添加事务到回退栈，前一个Fragment实例会被销毁。这里很明显，我们调用tx.addToBackStack(null)将当前的事务添加到了回退栈，所以FragmentOne实例不会被销毁，但是视图层次依然会被销毁，即会调用onDestoryView和onCreateView。所以【请注意】，当之后我们从FragmentTwo返回到前一个页面的时候，视图层仍旧是重新按照代码绘制，这里仅仅是是实例没有销毁，因此显示的页面中没有change几个字。

FragmentTwo.class文件

```
public class FragmentTwo extends Fragment implements OnClickList
ener {

    private Button mBtn ;

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup
container,
            Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.fragment_two, cont
ainer, false);
        mBtn = (Button) view.findViewById(R.id.id_fragment_two_b
tn);
        mBtn.setOnClickListener(this);
        return view ;
    }

    @Override
    public void onClick(View v)  {
        FragmentThree fThree = new FragmentThree();
        FragmentManager fm = getFragmentManager();
        FragmentTransaction tx = fm.beginTransaction();
        tx.hide(this);
        tx.add(R.id.id_content , fThree, "THREE");
        //tx.replace(R.id.id_content, fThree, "THREE");
        tx.addToBackStack(null);
        tx.commit();
    }

}
```

这里点击时，我们没有使用replace，而是先隐藏了当前的Fragment，然后添加了FragmentThree的实例，最后将事务添加到回退栈。这样做的目的是为了给大家提供一种方案：如果不希望视图重绘该怎么做，请再次仔细看效果图，我们在FragmentTwo的EditText填写的内容，用户点击返回键回来时，内容还在。

FragmentThree.class文件

```
public class FragmentThree extends Fragment implements OnClickLi
stener {

    private Button mBtn;

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup
container,
            Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.fragment_three, co
ntainer, false);
        mBtn = (Button) view.findViewById(R.id.id_fragment_three
_btn);
        mBtn.setOnClickListener(this);
        return view;
    }

    @Override
    public void onClick(View v) {
        Toast.makeText(getActivity(), " i am a btn in Fragment t
hree",
                Toast.LENGTH_SHORT).show();
    }

}
```

如果你还是不明白请仔细将上面的代码反复敲几遍

5. Fragment与Activity之间的通信

Fragment依附于Activity存在，因此与Activity之间的通信可以归纳为以下几点：

- [ ] 如果你Activity中包含自己管理的Fragment的引用，可以通过引用直接访问所有的Fragment的public方法
- [ ] 如果Activity中未保存任何Fragment的引用，那么没关系，每个Fragment都有一个唯一的TAG或者ID,可以通过

getFragmentManager.findFragmentByTag()或者findFragmentById()获得任何Fragment实例，然后进行操作
- [ ] Fragment中可以通过getActivity()得到当前绑定的Activity的实例，然后进行操作。

6. Fragment与Activity通信的优化

因为要考虑Fragment的重复使用，所以必须降低Fragment与Activity的耦合，而且Fragment更不应该直接操作别的Fragment，毕竟Fragment操作应该由它的管理者Activity来决定。

实现与上一个代码案例一模一样的功能与效果

FragmentOne.class文件

```
public class FragmentOne extends Fragment implements OnClickList
ener {

private Button mBtn;

    //设置按钮点击的回调
    public interface FOneBtnClickListener {
        void onFOneBtnClick();
    }

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup
container,
            Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.fragment_one, cont
ainer, false);
        mBtn = (Button) view.findViewById(R.id.id_fragment_one_b
tn);
        mBtn.setOnClickListener(this);
        return view;
    }

   //交给宿主Activity处理，如果它希望处理
    @Override
    public void onClick(View v) {
        if (getActivity() instanceof FOneBtnClickListener) {
            ((FOneBtnClickListener) getActivity()).onFOneBtnClic
k();
        }
    }

}
```

可以看到，现在的FragmentOne不和任何Activity耦合，任何Activity都可以使用，并且我们声明了一个接口，来回调其点击事件，想要重写其点击事件的Activity实现此接口即可，可以看到我们在onClick中首先判断了当前绑定的Activity是否实现了该接口，如果实现了则调用。

FragmentTwo.class文件

```
public class FragmentTwo extends Fragment implements OnClickList
ener {

    private Button mBtn ;
    private FTwoBtnClickListener fTwoBtnClickListener ;

    public interface FTwoBtnClickListener {
        void onFTwoBtnClick();
    }

    //设置回调接口
    public void setfTwoBtnClickListener(FTwoBtnClickListener fTw
oBtnClickListener) {
        this.fTwoBtnClickListener = fTwoBtnClickListener;
    }

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup
container,
            Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.fragment_two, cont
ainer, false);
        mBtn = (Button) view.findViewById(R.id.id_fragment_two_b
tn);
        mBtn.setOnClickListener(this);
        return view ;
    }

    @Override
    public void onClick(View v) {
        if(fTwoBtnClickListener != null)  {
            fTwoBtnClickListener.onFTwoBtnClick();
        }
    }

}
```

与FragmentOne极其类似，但是我们提供了setListener这样的方法，意味着Activity不仅需要实现该接口，还必须显示调用mFTwo.setfTwoBtnClickListener(this)。

MainActivity.class文件

```
public class MainActivity extends Activity implements FOneBtnCli
ckListener,
        FTwoBtnClickListener {

    private FragmentOne mFOne;
    private FragmentTwo mFTwo;
    private FragmentThree mFThree;  //FragmentThree代码参考上一个例
子中的代码

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        requestWindowFeature(Window.FEATURE_NO_TITLE);
        setContentView(R.layout.activity_main);

        mFOne = new FragmentOne();
        FragmentManager fm = getFragmentManager();
        FragmentTransaction tx = fm.beginTransaction();
        tx.add(R.id.id_content, mFOne, "ONE");
        tx.commit();
    }

    //FragmentOne 按钮点击时的回调
    @Override
    public void onFOneBtnClick() {
        if (mFTwo == null) {
            mFTwo = new FragmentTwo();
            mFTwo.setfTwoBtnClickListener(this);
        }
        FragmentManager fm = getFragmentManager();
        FragmentTransaction tx = fm.beginTransaction();
        tx.replace(R.id.id_content, mFTwo, "TWO");
        tx.addToBackStack(null);
        tx.commit();
    }

    //FragmentTwo按钮点击时的回调
    @Override
```

```
    public void onFTwoBtnClick() {
        if (mFThree == null) {
            mFThree = new FragmentThree();
        }
        FragmentManager fm = getFragmentManager();
        FragmentTransaction tx = fm.beginTransaction();
        tx.hide(mFTwo);
        tx.add(R.id.id_content, mFThree, "THREE");
        //tx.replace(R.id.id_content, fThree, "THREE");
        tx.addToBackStack(null);
        tx.commit();
    }

}
```

代码重构结束，与开始的效果一模一样。上面两种通信方式都是值得推荐的，随便选择一种自己喜欢的。这里再提一下：虽然Fragment和Activity可以通过getActivity与findFragmentByTag或者findFragmentById，进行任何操作，甚至在Fragment里面操作另外的Fragment，但是没有特殊理由是绝对不提倡的。Activity担任的是Fragment间类似总线一样的角色，应当由它决定Fragment如何操作。另外虽然Fragment不能响应Intent打开，但是Activity可以，Activity可以接收Intent，然后根据参数判断显示哪个Fragment。

7. 如何处理运行时配置发生变化

- 在Activity的学习中我们都知道，当屏幕旋转时，是对屏幕上的视图进行了重新绘制。因为当屏幕发生旋转，Activity发生重新启动，默认的Activity中的Fragment也会跟着Activity重新创建，用脚趾头都明白...横屏和竖屏显示的不一样肯定是进行了重新绘制视图的操作。所以，不断的旋转就不断绘制，这是一种很耗费内存资源的操作，那么如何来进行优化？

  代码分析：

  Fragment的class文件

```
public class FragmentOne extends Fragment {

    private static final String TAG = "FragmentOne";

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup
container,
            Bundle savedInstanceState) {
        Log.e(TAG, "onCreateView");
        View view = inflater.inflate(R.layout.fragment_one, cont
ainer, false);
        return view;
    }

    @Override
    public void onCreate(Bundle savedInstanceState) {
        // TODO Auto-generated method stub
        super.onCreate(savedInstanceState);
        Log.e(TAG, "onCreate");
    }

    @Override
    public void onDestroyView() {
        // TODO Auto-generated method stub
        super.onDestroyView();
        Log.e(TAG, "onDestroyView");
    }

    @Override
    public void onDestroy() {
        // TODO Auto-generated method stub
        super.onDestroy();
        Log.e(TAG, "onDestroy");
    }

}
```

然后你多次翻转屏幕都会打印如下log

```
07-20 08:18:46.651: E/FragmentOne(1633): onCreate
07-20 08:18:46.651: E/FragmentOne(1633): onCreate
07-20 08:18:46.651: E/FragmentOne(1633): onCreate
07-20 08:18:46.681: E/FragmentOne(1633): onCreateView
07-20 08:18:46.831: E/FragmentOne(1633): onCreateView
07-20 08:18:46.891: E/FragmentOne(1633): onCreateView
```

因为当屏幕发生旋转，Activity发生重新启动，默认的Activity中的Fragment也会跟着Activity重新创建；这样造成当旋转的时候，本身存在的Fragment会重新启动，然后当执行Activity的onCreate时，又会再次实例化一个新的Fragment，这就是出现的原因。

那么如何解决呢：

通过检查onCreate的参数Bundle savedInstanceState就可以判断，当前是否发生Activity的重新创建

默认的savedInstanceState会存储一些数据，包括Fragment的实例

所以，我们简单改一下代码，判断只有在savedInstanceState==null时，才进行创建Fragment实例

MainActivity.class文件

```
public class MainActivity extends Activity {

    private static final String TAG = "FragmentOne";
    private FragmentOne mFOne;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        requestWindowFeature(Window.FEATURE_NO_TITLE);
        setContentView(R.layout.activity_main);

        Log.e(TAG, savedInstanceState+"");

        if(savedInstanceState == null) {
            mFOne = new FragmentOne();
            FragmentManager fm = getFragmentManager();
            FragmentTransaction tx = fm.beginTransaction();
            tx.add(R.id.id_content, mFOne, "ONE");
            tx.commit();
        }
    }

}
```

现在无论进行多次旋转都只会有一个Fragment实例在Activity中，现在还存在一个问题，就是重新绘制时，Fragment发生重建，原本的数据如何保持？ 和Activity类似，Fragment也有onSaveInstanceState的方法，在此方法中进行保存数据，然后在onCreate或者onCreateView或者onActivityCreated进行恢复都可以。

一、消息机制概述

1.消息机制的简介

在Android中使用消息机制，我们首先想到的就是Handler。没错，Handler是Android消息机制的上层接口。Handler的使用过程很简单，通过它可以轻松地将一个任务切换到Handler所在的线程中去执行。通常情况下，Handler的使用场景就是更新UI。

如下就是使用消息机制的一个简单实例：

```
public class Activity extends android.app.Activity {
    private Handler mHandler = new Handler(){
        @Override
        public void handleMessage(Message msg) {
            super.handleMessage(msg);
            System.out.println(msg.what);
        }
    };
    @Override
    public void onCreate(Bundle savedInstanceState, PersistableB
undle persistentState) {
        super.onCreate(savedInstanceState, persistentState);
        setContentView(R.layout.activity_main);
        new Thread(new Runnable() {
            @Override
            public void run() {
                ...............耗时操作
                Message message = Message.obtain();
                message.what = 1;
                mHandler.sendMessage(message);
            }
        }).start();
    }
}
```

在子线程中，进行耗时操作，执行完操作后，发送消息，通知主线程更新UI。这便是消息机制的典型应用场景。我们通常只会接触到Handler和Message来完成消息机制，其实内部还有两大助手来共同完成消息传递。

2.消息机制的模型

消息机制主要包含：MessageQueue，Handler和Looper这三大部分，以及Message，下面我们一一介绍。

Message：需要传递的消息，可以传递数据；

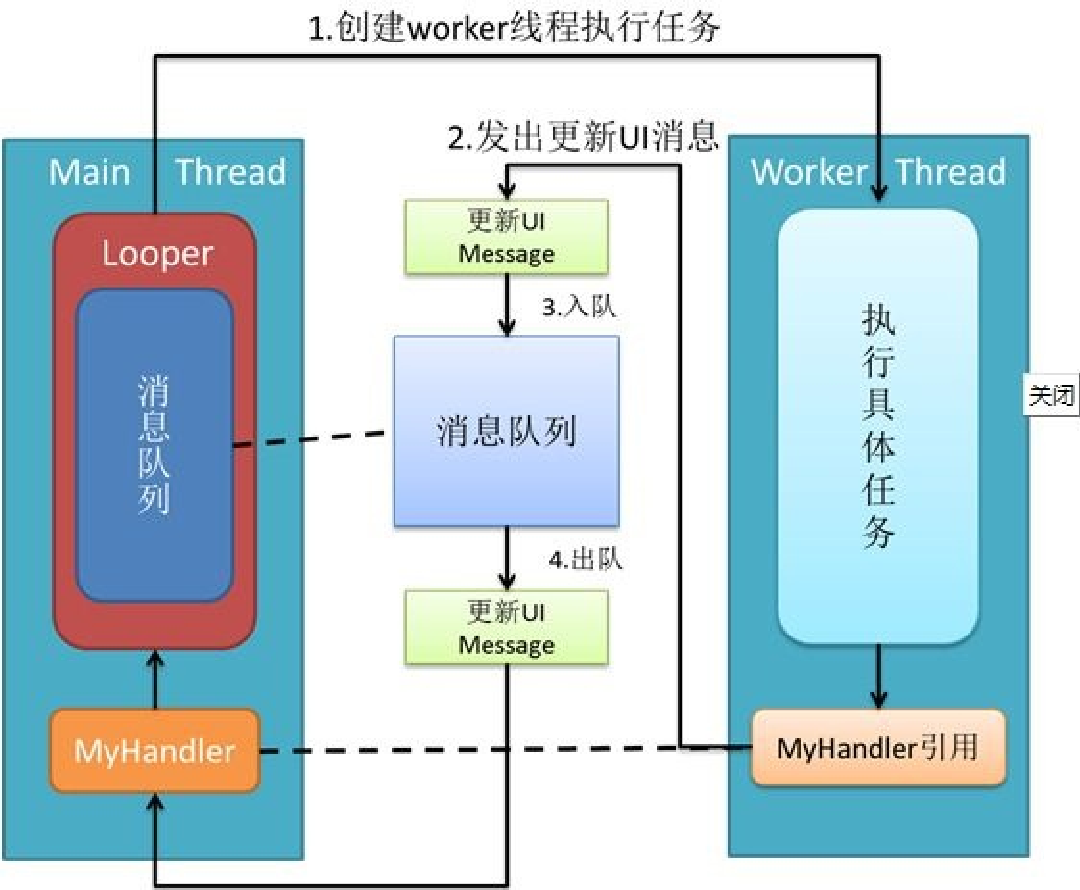
MessageQueue：消息队列，但是它的内部实现并不是用的队列，实际上是通过一个单链表的数据结构来维护消息列表，因为单链表在插入和删除上比较有优势。主要功能向消息池投递消息(MessageQueue.enqueueMessage)和取走消息池的消息(MessageQueue.next)；

Handler：消息辅助类，主要功能向消息池发送各种消息事件(Handler.sendMessage)和处理相应消息事件(Handler.handleMessage)；

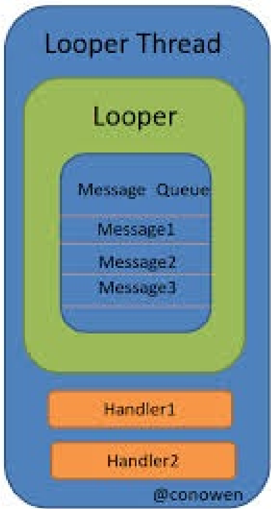
Looper：不断循环执行(Looper.loop)，从MessageQueue中读取消息，按分发机制将消息分发给目标处理者。

3.消息机制的架构

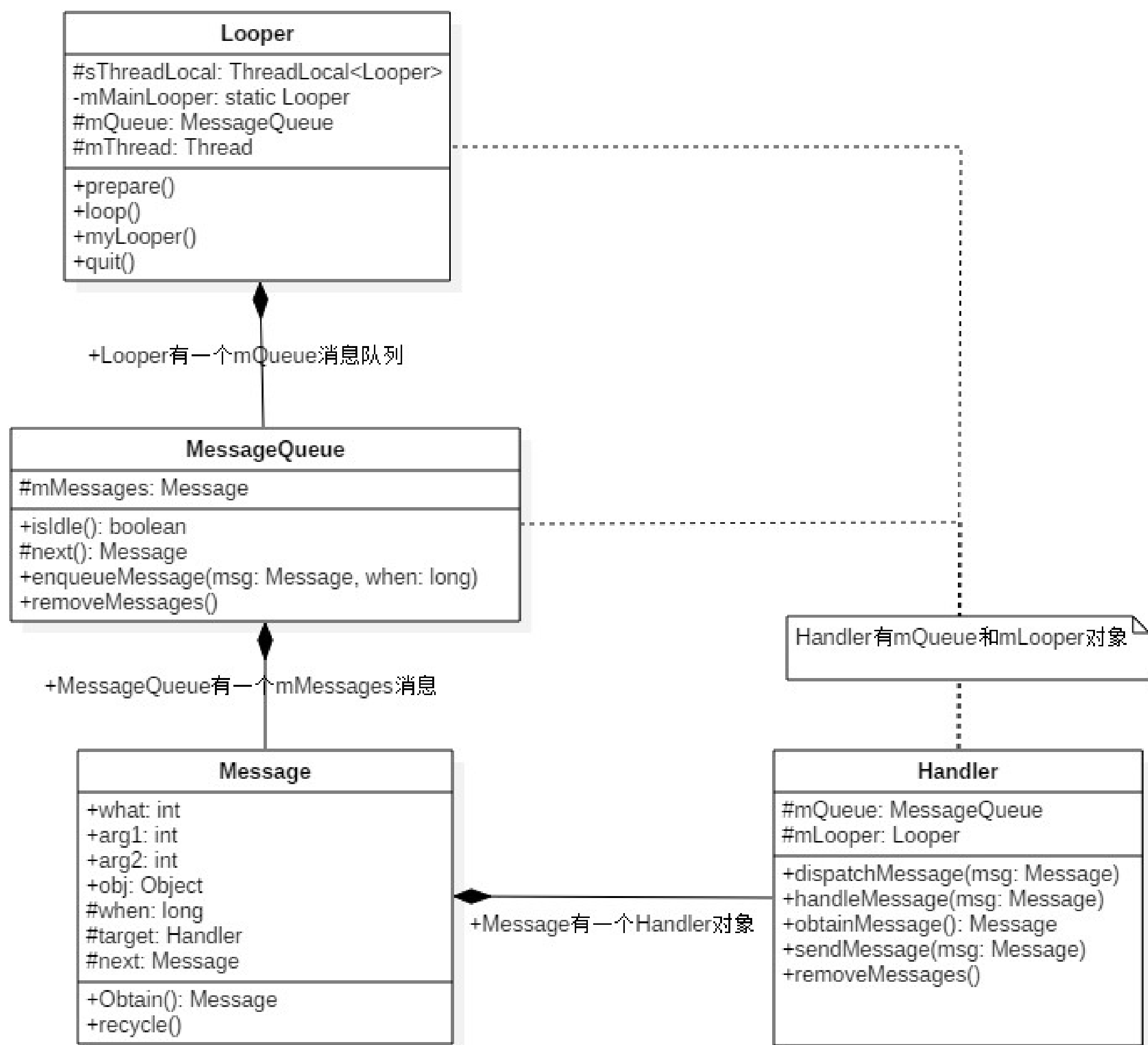
消息机制的运行流程：在子线程执行完耗时操作，当Handler发送消息时，将会调用 `MessageQueue.enqueueMessage` ，向消息队列中添加消息。当通过 `Looper.loop` 开启循环后，会不断地从线程池中读取消息，即调用 `MessageQueue.next` ，然后调用目标Handler（即发送该消息的Handler）的 `dispatchMessage` 方法传递消息，然后返回到Handler所在线程，目标Handler收到消息，调用 `handleMessage` 方法，接收消息，处理消息。



MessageQueue，**Handler**和**Looper**三者之间的关系：每个线程中只能存在一个Looper，Looper是保存在ThreadLocal中的。主线程（UI线程）已经创建了一个Looper，所以在主线程中不需要再创建Looper，但是在其他线程中需要创建Looper。每个线程中可以有多个Handler，即一个Looper可以处理来自多个Handler的消息。 Looper中维护一个MessageQueue，来维护消息队列，消息队列中的Message可以来自不同的Handler。



下面是消息机制的整体架构图，接下来我们将慢慢解剖整个架构。



从中我们可以看出：
Looper有一个MessageQueue消息队列；
MessageQueue有一组待处理的Message；
Message中记录发送和处理消息的Handler；
Handler中有Looper和MessageQueue。

二、消息机制的源码解析

1.Looper

要想使用消息机制，首先要创建一个Looper。
初始化**Looper**
无参情况下，默认调用 `prepare(true);` 表示的是这个Looper可以退出，而对于false的情况则表示当前Looper不可以退出。。

```
 public static void prepare() {
        prepare(true);
    }

    private static void prepare(boolean quitAllowed) {
        if (sThreadLocal.get() != null) {
            throw new RuntimeException("Only one Looper may be c
reated per thread");
        }
        sThreadLocal.set(new Looper(quitAllowed));
    }
```

这里看出，不能重复创建Looper，只能创建一个。创建Looper,并保存在ThreadLocal。其中ThreadLocal是线程本地存储区（Thread Local Storage，简称为TLS），每个线程都有自己的私有的本地存储区域，不同线程之间彼此不能访问对方的TLS区域。

开启**Looper**

```
public static void loop() {
    final Looper me = myLooper();  //获取TLS存储的Looper对象
    if (me == null) {
        throw new RuntimeException("No Looper; Looper.prepare()
wasn't called on this thread.");
    }
    final MessageQueue queue = me.mQueue;  //获取Looper对象中的消息
队列

    Binder.clearCallingIdentity();
    final long ident = Binder.clearCallingIdentity();

    for (;;) { //进入loop的主循环方法
        Message msg = queue.next(); //可能会阻塞,因为next()方法可能
会无限循环
        if (msg == null) { //消息为空，则退出循环
            return;
        }

        Printer logging = me.mLogging;  //默认为null，可通过setMes
```

```
sageLogging()方法来指定输出，用于debug功能
        if (logging != null) {
            logging.println(">>>>> Dispatching to " + msg.target
 + " " +
                    msg.callback + ": " + msg.what);
        }
        msg.target.dispatchMessage(msg); //获取msg的目标Handler，
然后用于分发Message
        if (logging != null) {
            logging.println("<<<<< Finished to " + msg.target +
" " + msg.callback);
        }

        final long newIdent = Binder.clearCallingIdentity();
        if (ident != newIdent) {

        }
        msg.recycleUnchecked();
    }
}
```

loop()进入循环模式，不断重复下面的操作，直到消息为空时退出循环:
读取MessageQueue的下一条Message（关于next()，后面详细介绍）；
把Message分发给相应的target。

当**next()**取出下一条消息时，队列中已经没有消息时，**next()**会无限循环，产生阻塞。等待**MessageQueue**中加入消息，然后重新唤醒。

主线程中不需要自己创建**Looper**，这是由于在程序启动的时候，系统已经帮我们自动调用了 `Looper.prepare()` 方法。查看**ActivityThread**中的 `main()` 方法，代码如下所示：

```
  public static void main(String[] args) {
.........................
        Looper.prepareMainLooper();
  .........................
        Looper.loop();
  .........................

    }
```

其中 `` `prepareMainLooper() `` 方法会调用 `prepare(false)` 方法。

2.Handler

创建Handler

```
public Handler() {
    this(null, false);
}

public Handler(Callback callback, boolean async) {
   .................................
    //必须先执行Looper.prepare()，才能获取Looper对象，否则为null.
    mLooper = Looper.myLooper();  //从当前线程的TLS中获取Looper对象
    if (mLooper == null) {
        throw new RuntimeException("");
    }
    mQueue = mLooper.mQueue; //消息队列，来自Looper对象
    mCallback = callback;  //回调方法
    mAsynchronous = async; //设置消息是否为异步处理方式
}
```

对于Handler的无参构造方法，默认采用当前线程TLS中的Looper对象，并且callback回调方法为null，且消息为同步处理方式。只要执行的 `Looper.prepare()` 方法，那么便可以获取有效的Looper对象。

3.发送消息

发送消息有几种方式，但是归根结底都是调用了 `sendMessageAtTime()` 方法。

在子线程中通过Handler的post()方式或send()方式发送消息，最终都是调用了 `sendMessageAtTime()` 方法。

post方法

```
 public final boolean post(Runnable r)
    {
       return  sendMessageDelayed(getPostMessage(r), 0);
    }
public final boolean postAtTime(Runnable r, long uptimeMillis)
    {
        return sendMessageAtTime(getPostMessage(r), uptimeMillis
);
    }
 public final boolean postAtTime(Runnable r, Object token, long
uptimeMillis)
    {
        return sendMessageAtTime(getPostMessage(r, token), uptim
eMillis);
    }
 public final boolean postDelayed(Runnable r, long delayMillis)
    {
        return sendMessageDelayed(getPostMessage(r), delayMillis
);
    }
```

send方法

```
public final boolean sendMessage(Message msg)
    {
        return sendMessageDelayed(msg, 0);
    }
 public final boolean sendEmptyMessage(int what)
    {
        return sendEmptyMessageDelayed(what, 0);
    }
public final boolean sendEmptyMessageDelayed(int what, long dela
yMillis) {
        Message msg = Message.obtain();
        msg.what = what;
        return sendMessageDelayed(msg, delayMillis);
    }
 public final boolean sendEmptyMessageAtTime(int what, long upti
meMillis) {
        Message msg = Message.obtain();
        msg.what = what;
        return sendMessageAtTime(msg, uptimeMillis);
    }
 public final boolean sendMessageDelayed(Message msg, long delay
Millis)
    {
        if (delayMillis < 0) {
            delayMillis = 0;
        }
        return sendMessageAtTime(msg, SystemClock.uptimeMillis()
 + delayMillis);
    }
```

就连子线程中调用Activity中的runOnUiThread()中更新UI，其实也是发送消息通知主线程更新UI，最终也会调用 `sendMessageAtTime()` 方法。

```
public final void runOnUiThread(Runnable action) {
        if (Thread.currentThread() != mUiThread) {
            mHandler.post(action);
        } else {
            action.run();
        }
    }
```

如果当前的线程不等于UI线程(主线程)，就去调用Handler的post()方法，最终会调用 `sendMessageAtTime()` 方法。否则就直接调用Runnable对象的run()方法。

下面我们就来一探究竟，到底 `sendMessageAtTime()` 方法有什么作用？

sendMessageAtTime()

```
public boolean sendMessageAtTime(Message msg, long uptimeMillis)
{
       //其中mQueue是消息队列，从Looper中获取的
        MessageQueue queue = mQueue;
        if (queue == null) {
            RuntimeException e = new RuntimeException(
                    this + " sendMessageAtTime() called with no
mQueue");
            Log.w("Looper", e.getMessage(), e);
            return false;
        }
        //调用enqueueMessage方法
        return enqueueMessage(queue, msg, uptimeMillis);
    }
```

```
 private boolean enqueueMessage(MessageQueue queue, Message msg,
long uptimeMillis) {
        msg.target = this;
        if (mAsynchronous) {
            msg.setAsynchronous(true);
        }
        //调用MessageQueue的enqueueMessage方法
        return queue.enqueueMessage(msg, uptimeMillis);
    }
```

可以看到sendMessageAtTime()`方法的作用很简单，就是调用MessageQueue的enqueueMessage()方法，往消息队列中添加一个消息。
下面来看enqueueMessage()方法的具体执行逻辑。

enqueueMessage()

```
boolean enqueueMessage(Message msg, long when) {
    // 每一个Message必须有一个target
    if (msg.target == null) {
        throw new IllegalArgumentException("Message must have a
target.");
    }
    if (msg.isInUse()) {
        throw new IllegalStateException(msg + " This message is
already in use.");
    }
    synchronized (this) {
        if (mQuitting) {  //正在退出时，回收msg，加入到消息池
            msg.recycle();
            return false;
        }
        msg.markInUse();
        msg.when = when;
        Message p = mMessages;
        boolean needWake;
        if (p == null || when == 0 || when < p.when) {
            //p为null(代表MessageQueue没有消息） 或者msg的触发时间是队
列中最早的， 则进入该该分支
            msg.next = p;
```

```
                mMessages = msg;
                needWake = mBlocked;
            } else {
                //将消息按时间顺序插入到MessageQueue。一般地，不需要唤醒事件
队列，除非
                //消息队头存在barrier，并且同时Message是队列中最早的异步消
息。
                needWake = mBlocked && p.target == null && msg.isAsy
nchronous();
                Message prev;
                for (;;) {
                    prev = p;
                    p = p.next;
                    if (p == null || when < p.when) {
                        break;
                    }
                    if (needWake && p.isAsynchronous()) {
                        needWake = false;
                    }
                }
                msg.next = p;
                prev.next = msg;
            }
            if (needWake) {
                nativeWake(mPtr);
            }
        }
        return true;
}
```

MessageQueue是按照Message触发时间的先后顺序排列的，队头的消息是将要最早触发的消息。当有消息需要加入消息队列时，会从队列头开始遍历，直到找到消息应该插入的合适位置，以保证所有消息的时间顺序。

4.获取消息

当发送了消息后，在MessageQueue维护了消息队列，然后在Looper中通过 `loop()` 方法，不断地获取消息。上面对 `loop()` 方法进行了介绍，其中最重要的是调用了 `queue.next()` 方法,通过该方法来提取下一条信息。下面我们来看

一下 `next()` 方法的具体流程。

next()

```
Message next() {
    final long ptr = mPtr;
    if (ptr == 0) { //当消息循环已经退出，则直接返回
        return null;
    }
    int pendingIdleHandlerCount = -1; // 循环迭代的首次为-1
    int nextPollTimeoutMillis = 0;
    for (;;) {
        if (nextPollTimeoutMillis != 0) {
            Binder.flushPendingCommands();
        }
        //阻塞操作，当等待nextPollTimeoutMillis时长，或者消息队列被唤醒
，都会返回
        nativePollOnce(ptr, nextPollTimeoutMillis);
        synchronized (this) {
            final long now = SystemClock.uptimeMillis();
            Message prevMsg = null;
            Message msg = mMessages;
            if (msg != null && msg.target == null) {
                //当消息Handler为空时，查询MessageQueue中的下一条异步
消息msg，为空则退出循环。
                do {
                    prevMsg = msg;
                    msg = msg.next;
                } while (msg != null && !msg.isAsynchronous());
            }
            if (msg != null) {
                if (now < msg.when) {
                    //当异步消息触发时间大于当前时间，则设置下一次轮询的
超时时长
                    nextPollTimeoutMillis = (int) Math.min(msg.w
hen - now, Integer.MAX_VALUE);
                } else {
                    // 获取一条消息，并返回
                    mBlocked = false;
                    if (prevMsg != null) {
                        prevMsg.next = msg.next;
```

```
                    } else {
                        mMessages = msg.next;
                    }
                    msg.next = null;
                    //设置消息的使用状态，即flags |= FLAG_IN_USE
                    msg.markInUse();
                    return msg;   //成功地获取MessageQueue中的下一
条即将要执行的消息
                }
            } else {
                //没有消息
                nextPollTimeoutMillis = -1;
            }
        //消息正在退出，返回null
            if (mQuitting) {
                dispose();
                return null;
            }
            ..............................
    }
}
```

nativePollOnce是阻塞操作，其中nextPollTimeoutMillis代表下一个消息到来前，还需要等待的时长；当nextPollTimeoutMillis = -1时，表示消息队列中无消息，会一直等待下去。

可以看出 `next()` 方法根据消息的触发时间，获取下一条需要执行的消息,队列中消息为空时，则会进行阻塞操作。

5.分发消息

在loop()方法中，获取到下一条消息后，执行 `msg.target.dispatchMessage(msg)` ，来分发消息到目标Handler对象。

下面就来具体看下 `dispatchMessage(msg)` 方法的执行流程。

dispatchMessage()

```
public void dispatchMessage(Message msg) {
    if (msg.callback != null) {
        //当Message存在回调方法，回调msg.callback.run()方法；
        handleCallback(msg);
    } else {
        if (mCallback != null) {
            //当Handler存在Callback成员变量时，回调方法handleMessage
();
            if (mCallback.handleMessage(msg)) {
                return;
            }
        }
        //Handler自身的回调方法handleMessage()
        handleMessage(msg);
    }
}
```

```
private static void handleCallback(Message message) {
        message.callback.run();
    }
```

分发消息流程：

当Message的 `msg.callback` 不为空时，则回调方法 `msg.callback.run()`；
当Handler的 `mCallback` 不为空时，则回调方法 `mCallback.handleMessage(msg)`；
最后调用Handler自身的回调方法 `handleMessage()`，该方法默认为空，Handler子类通过覆写该方法来完成具体的逻辑。

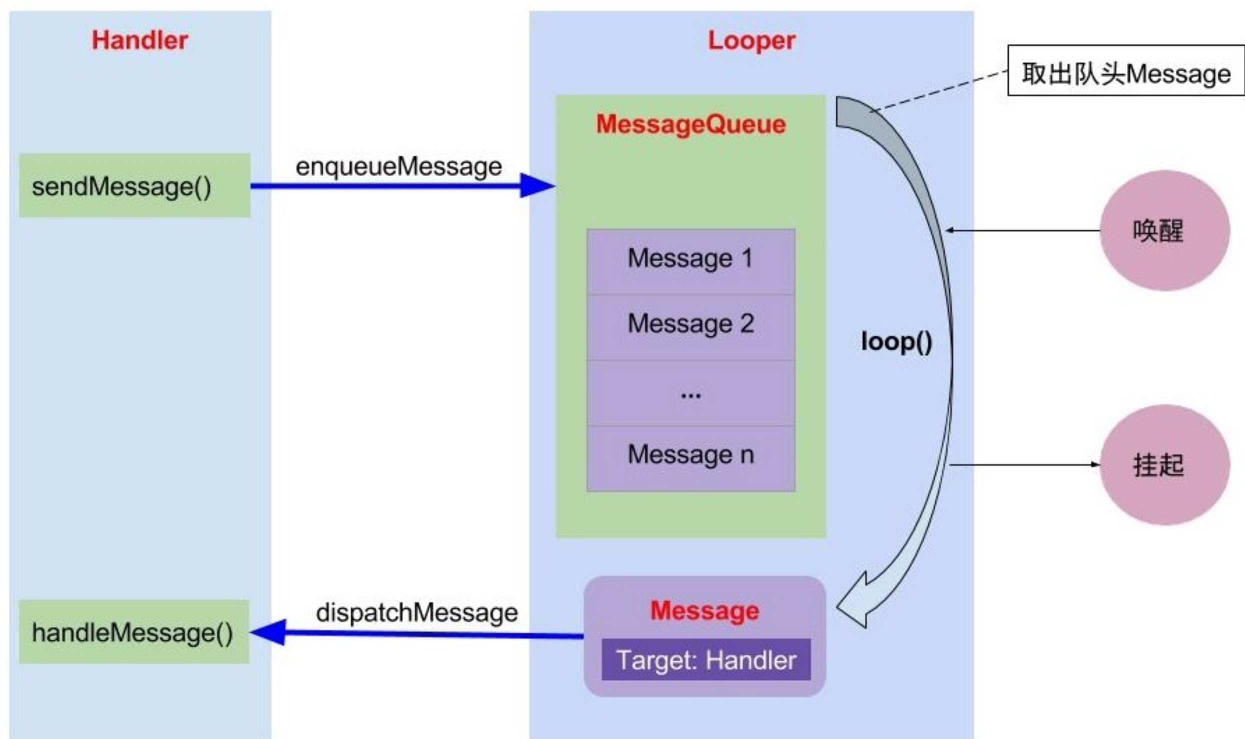
消息分发的优先级：

Message的回调方法：`message.callback.run()`，优先级最高；
Handler中Callback的回调方法：`Handler.mCallback.handleMessage(msg)`，优先级仅次于1；
Handler的默认方法：`Handler.handleMessage(msg)`，优先级最低。

对于很多情况下，消息分发后的处理方法是第3种情况，即 `Handler.handleMessage()`，一般地往往通过覆写该方法从而实现自己的业务逻辑。

三、总结

以上便是消息机制的原理，以及从源码角度来解析消息机制的运行过程。可以简单地用下图来理解。



一、 基础认知

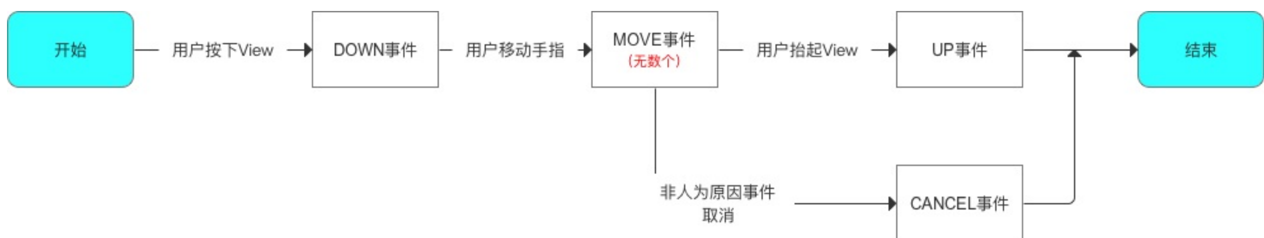
1.1 事件分发的对象是谁？

答：事件

- 当用户触摸屏幕时（View或ViewGroup派生的控件），将产生点击事件（Touch事件）。

  > Touch事件相关细节（发生触摸的位置、时间、历史记录、手势动作等）被封装成MotionEvent对象

- 主要发生的Touch事件有如下四种：
  - MotionEvent.ACTION_DOWN：按下View（所有事件的开始）
  - MotionEvent.ACTION_MOVE：滑动View
  - MotionEvent.ACTION_CANCEL：非人为原因结束本次事件
  - MotionEvent.ACTION_UP：抬起View（与DOWN对应）
- 事件列：从手指接触屏幕至手指离开屏幕，这个过程产生的一系列事件 任何事件列都是以DOWN事件开始，UP事件结束，中间有无数的MOVE事件，如下图：



即当一个MotionEvent 产生后，系统需要把这个事件传递给一个具体的 View 去处理,

1.2 事件分发的本质

答：将点击事件（**MotionEvent**）向某个**View**进行传递并最终得到处理

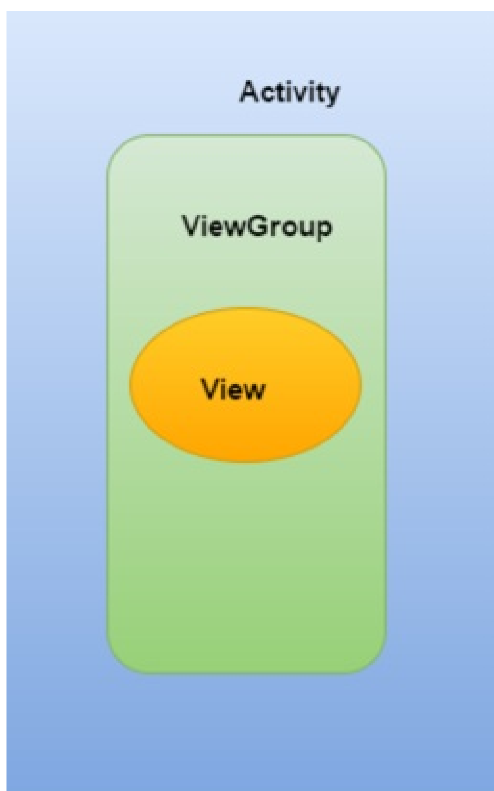
> 即当一个点击事件发生后，系统需要将这个事件传递给一个具体的View去处理。这个事件传递的过程就是分发过程。

1.3 事件在哪些对象之间进行传递？

答：**Activity、ViewGroup、View**

> 一个点击事件产生后，传递顺序是：Activity（Window） -> ViewGroup -> View

- Android的UI界面是由Activity、ViewGroup、View及其派生类组合而成的



- View是所有UI组件的基类

> 一般Button、ImageView、TextView等控件都是继承父类View

- ViewGroup是容纳UI组件的容器，即一组View的集合（包含很多子View和子VewGroup），

> 1. 其本身也是从View派生的，即ViewGroup是View的子类
> 2. 是Android所有布局的父类或间接父类：项目用到的布局（LinearLayout、RelativeLayout等），都继承自ViewGroup，即属于ViewGroup子类。
> 3. 与普通View的区别：ViewGroup实际上也是一个View，只不过比起View，它多了可以包含子View和定义布局参数的功能。

1.4 事件分发过程由哪些方法协作完成？

答：**dispatchTouchEvent() 、onInterceptTouchEvent()**和**onTouchEvent()**

方法	作用	调用时刻
dispatchTouchEvent()	分发(传递)点击事件	当点击事件能够传递给当前View时，该方法就会被调用
onInterceptTouchEvent()	判断是否拦截了某个事件（只存在于ViewGroup，普通的View没有这个方法）	在dispatchTouchEvent()内部调用；
onTouchEvent()	处理点击事件	在dispatchTouchEvent()内部调用；

> 下文会对这3个方法进行详细介绍

1.5 总结

- Android事件分发机制的本质是要解决：

  点击事件由哪个对象发出，经过哪些对象，最终达到哪个对象并最终得到处理。

  > 这里的对象是指Activity、ViewGroup、View

- Android中事件分发顺序：**Activity（Window） -> ViewGroup -> View**
- 事件分发过程由dispatchTouchEvent() 、onInterceptTouchEvent()和onTouchEvent()三个方法协助完成

经过上述3个问题，相信大家已经对Android的事件分发有了感性的认知，接下来，我将详细介绍Android事件分发机制。

二、 事件分发机制方法&流程介绍

- 事件分发过程由dispatchTouchEvent() 、onInterceptTouchEvent()和

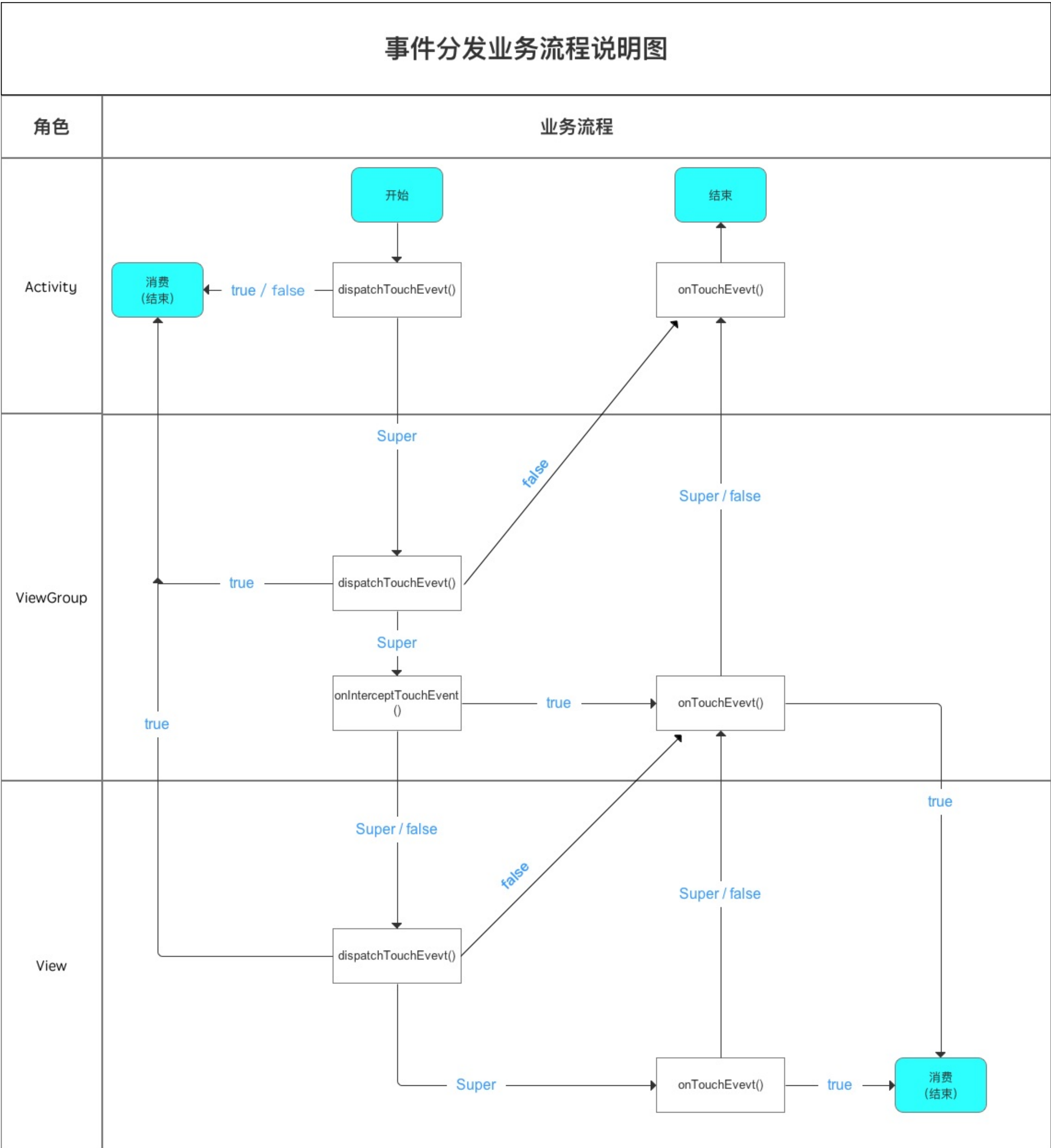
onTouchEvent()三个方法协助完成，如下图：

方法	作用	调用时刻	返回结果		
dispatchTouchEvent()	分发(传递)点击事件	当点击事件能够传递给当前View时，该方法就会被调用	是否消费当前事件	默认实现	根据当前对象的不同而返回方法不同 Activity：super.dispatchTouchEvent()，即调用父类ViewGroup的dispatchTouchEvent() ViewGroup：onIntercepTouchEvent()，即调用自身的onIntercepTouchEvent() View：onTouchEvent （），即调用自身的onTouchEvent （）
				true	• 消费事件，且事件不会往下传递； • 后续事件会继续分发到该 View
				false	• 不消费事件，且事件停止传递； • 将事件回传给父控件的onTouchEvent()处理 • 当前View仍然接受此事件的其他事件(与onTouchEvent()区别)
onInterceptTouchEvent()	判断是否拦截了某个事件（只存在于ViewGroup，普通的View没有这个方法）	在ViewGroup的dispatchTouchEvent()内部调用；	是否拦截当前事件；	true	• 拦截事件，且事件不会往下传递； • 自己处理事件，即执行自己的onTouchEvent()； • 同一个事件列的其他事件都直接交由该View处理；在同一个事件列中该方法不会再次被调用；
				false（默认）	• 不拦截事件，且事件会继续往下传递； • 事件传递到子 view， 调用父类View.dispatchTouchEvent() 方法中去处理 • 当前View仍然接受此事件的其他事件 （与onTouchEvent()区别）
onTouchEvent()	处理点击事件	在dispatchTouchEvent()内部调用；	是否消费（处理）当前事件	true	• 自己处理事件，且事件不会往下传递； • 后续事件序列让其处理；
				false	• 不处理事件； • 将事件向上传给父控件的onTouchEvent()处理 • 当前View不再接受此事件的其他事件(与dispatchTouchEvent()、onInterceptTouchEvent()区别)

方法详细介绍

- Android事件分发流程如下：（必须熟记）

> Android事件分发顺序：**Activity（Window） -> ViewGroup -> View**



其中：

- super：调用父类方法
- true：消费事件，即事件不继续往下传递
- false：不消费事件，事件也不继续往下传递 / 交由给父控件onTouchEvent（）处理

接下来，我将详细介绍这**3**个方法及相关流程。

2.1 dispatchTouchEvent()

属性	介绍
使用对象	Activity、ViewGroup、View
作用	分发点击事件
调用时刻	当点击事件能够传递给当前View时，该方法就会被调用
返回结果	是否消费当前事件，详细情况如下：

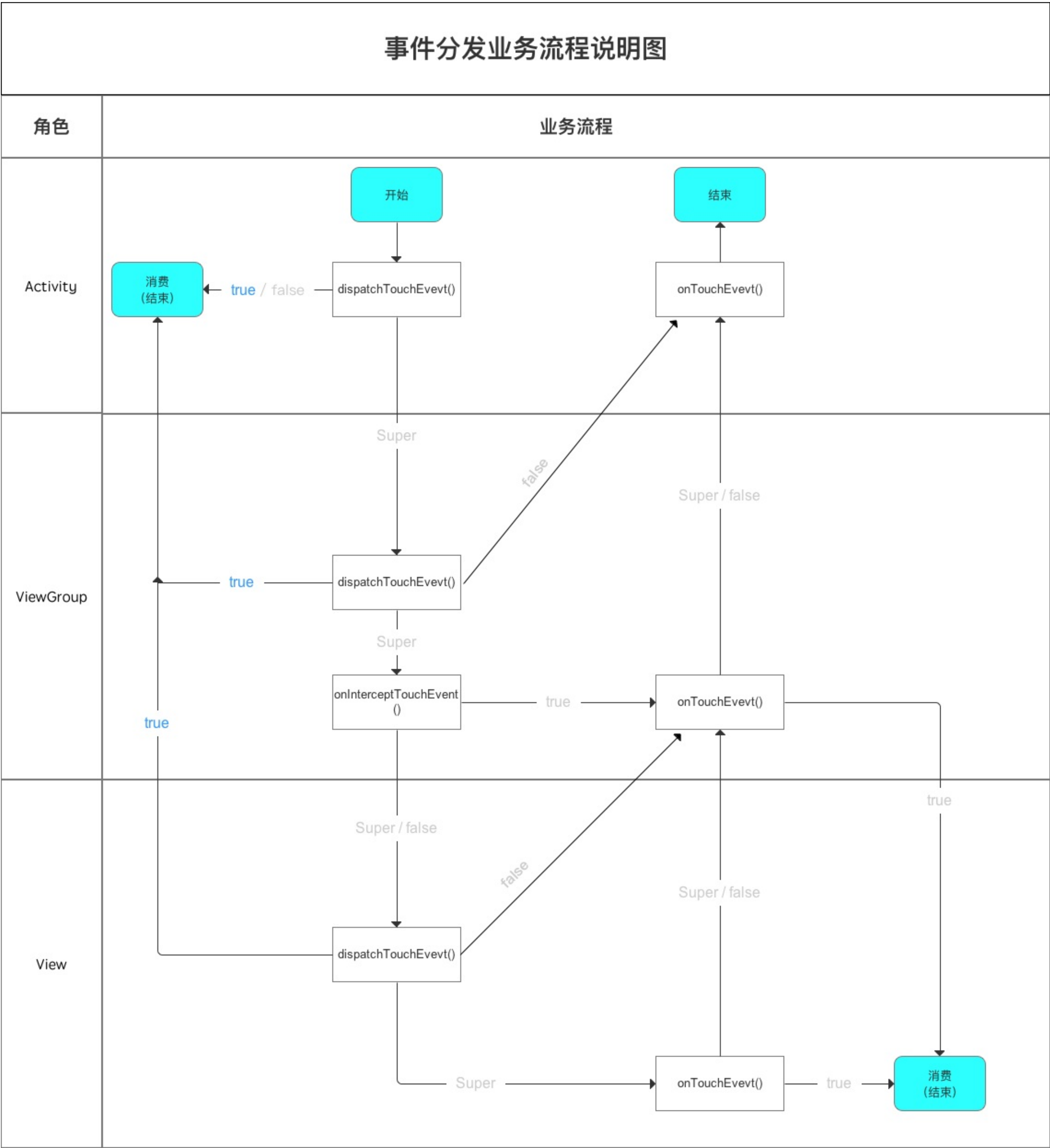
1. 默认情况：根据当前对象的不同而返回方法不同

对象	返回方法	备注
Activity	super.dispatchTouchEvent()	即调用父类ViewGroup的 dispatchTouchEvent()
ViewGroup	onIntercepTouchEvent()	即调用自身的 onIntercepTouchEvent()
View	onTouchEvent（）	即调用自身的onTouchEvent（）

流程解析

2. 返回**true**

- 消费事件
- 事件不会往下传递
- 后续事件（Move、Up）会继续分发到该View
- 流程图如下：

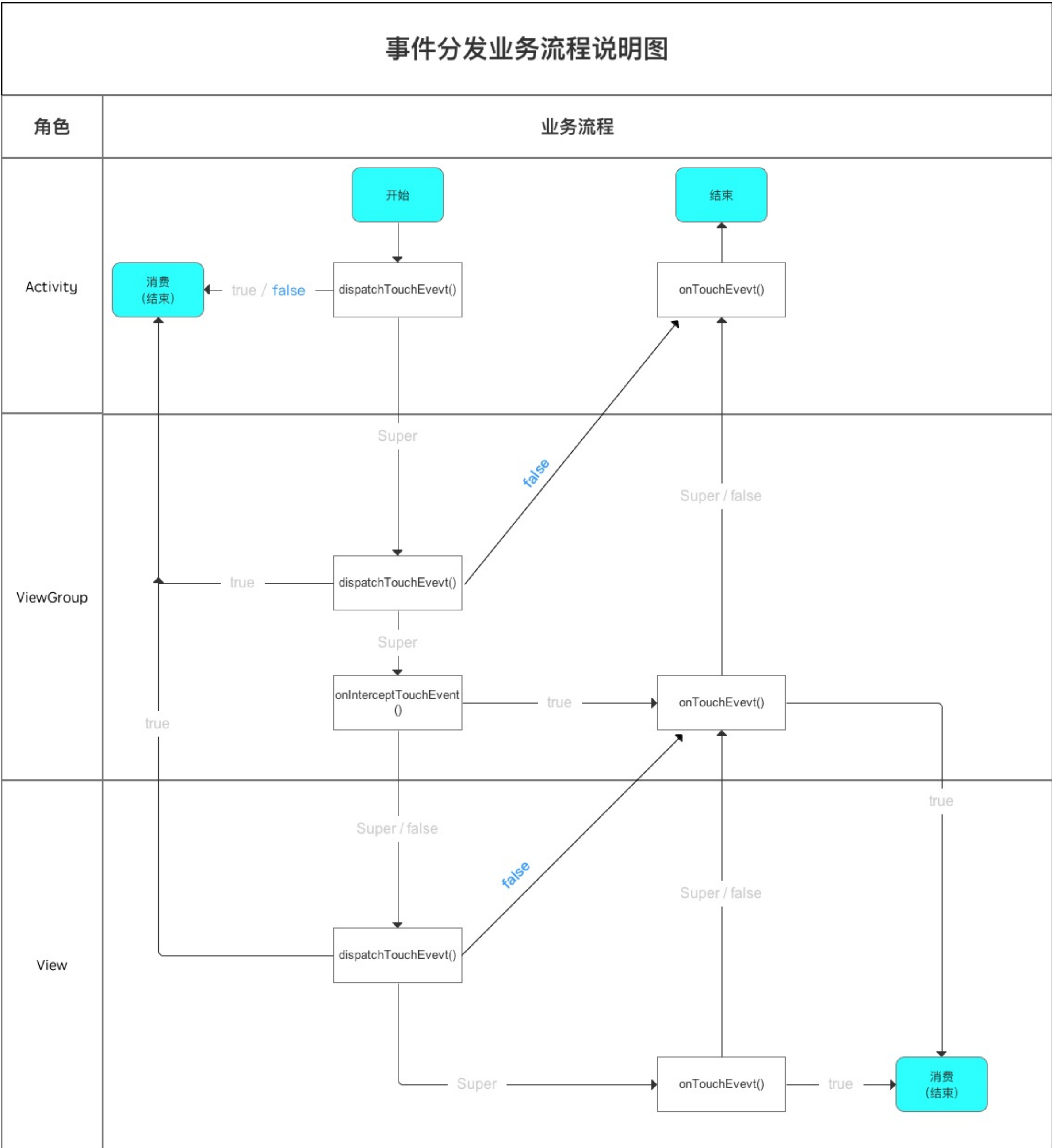


3. 返回false

- 不消费事件
- 事件不会往下传递
- 将事件回传给父控件的onTouchEvent()处理

  > Activity例外：返回false=消费事件

- 后续事件（Move、Up）会继续分发到该View(与onTouchEvent()区别）
- 流程图如下：



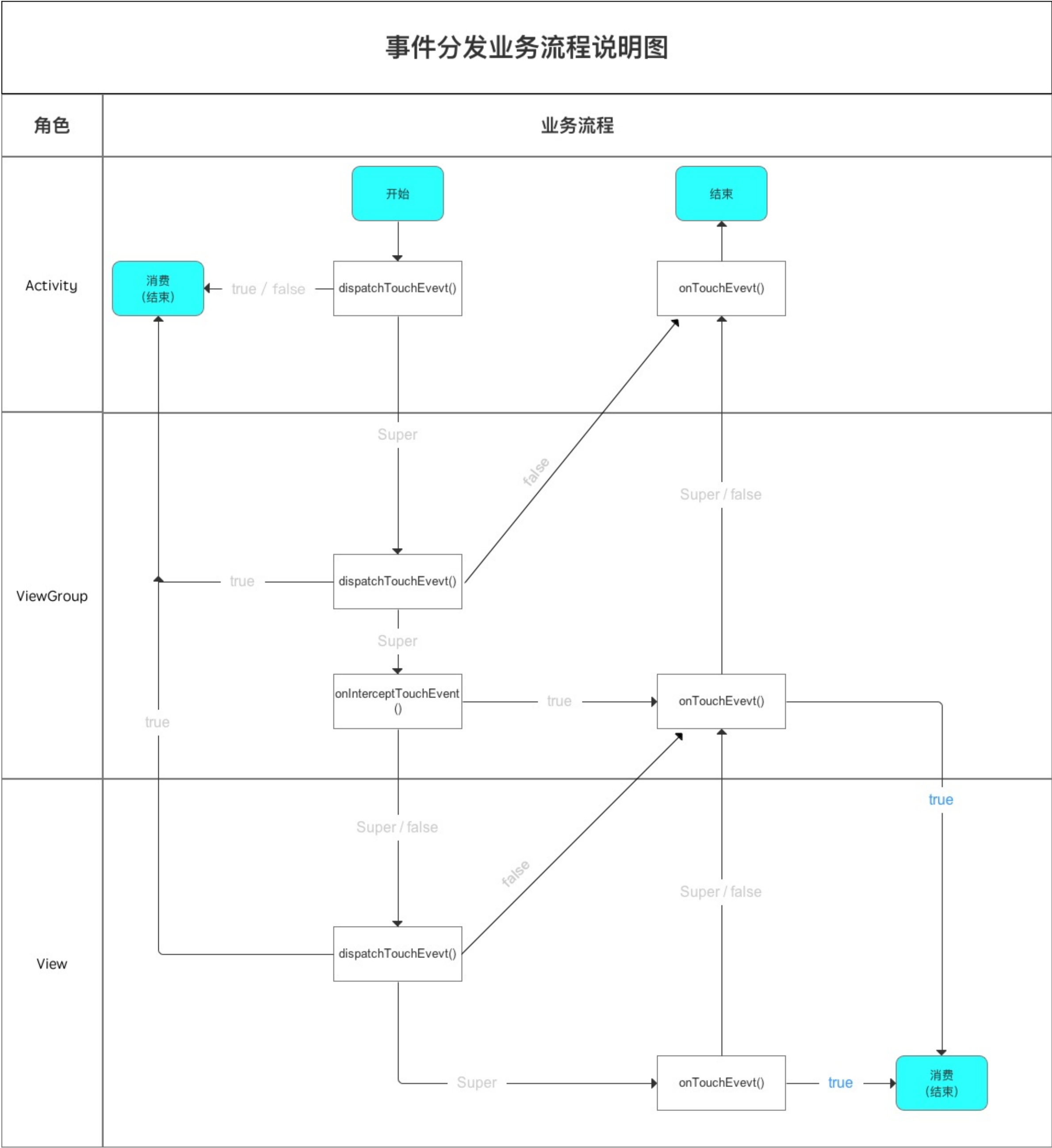
2.2 onTouchEvent()

属性	介绍
使用对象	Activity、ViewGroup、View
作用	处理点击事件
调用时刻	在dispatchTouchEvent()内部调用
返回结果	是否消费（处理）当前事件，详细情况如下：

> 与dispatchTouchEvent()类似

1. 返回true

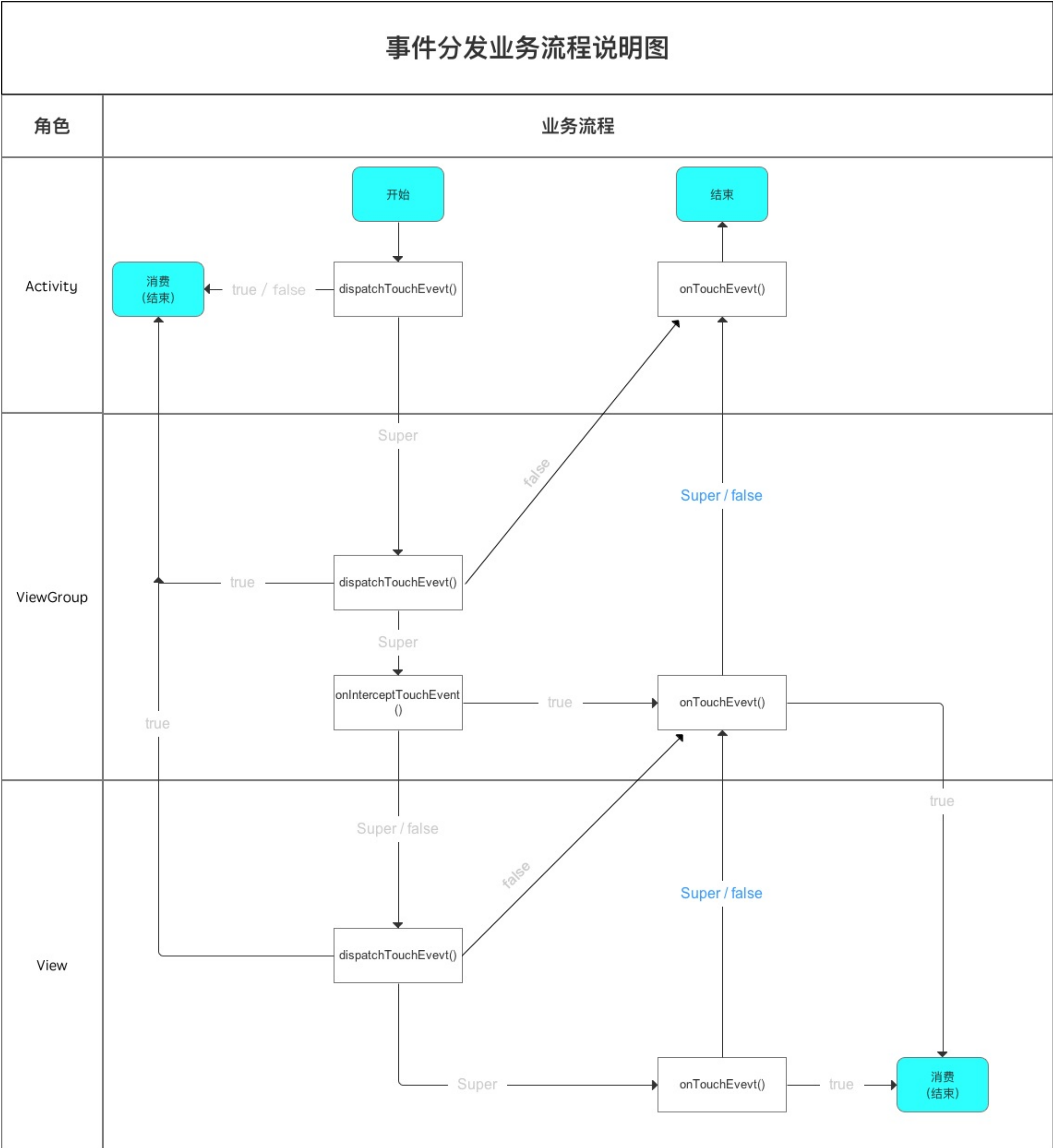
- 自己处理（消费）该事情
- 事件停止传递
- 该事件序列的后续事件（Move、Up）让其处理；
- 流程图如下：



2. 返回false（同默认实现：调用父类onTouchEvent()）

- 不处理（消费）该事件
- 事件往上传递给父控件的onTouchEvent()处理

- 当前View不再接受此事件列的其他事件（Move、Up）；
- 流程图如下：



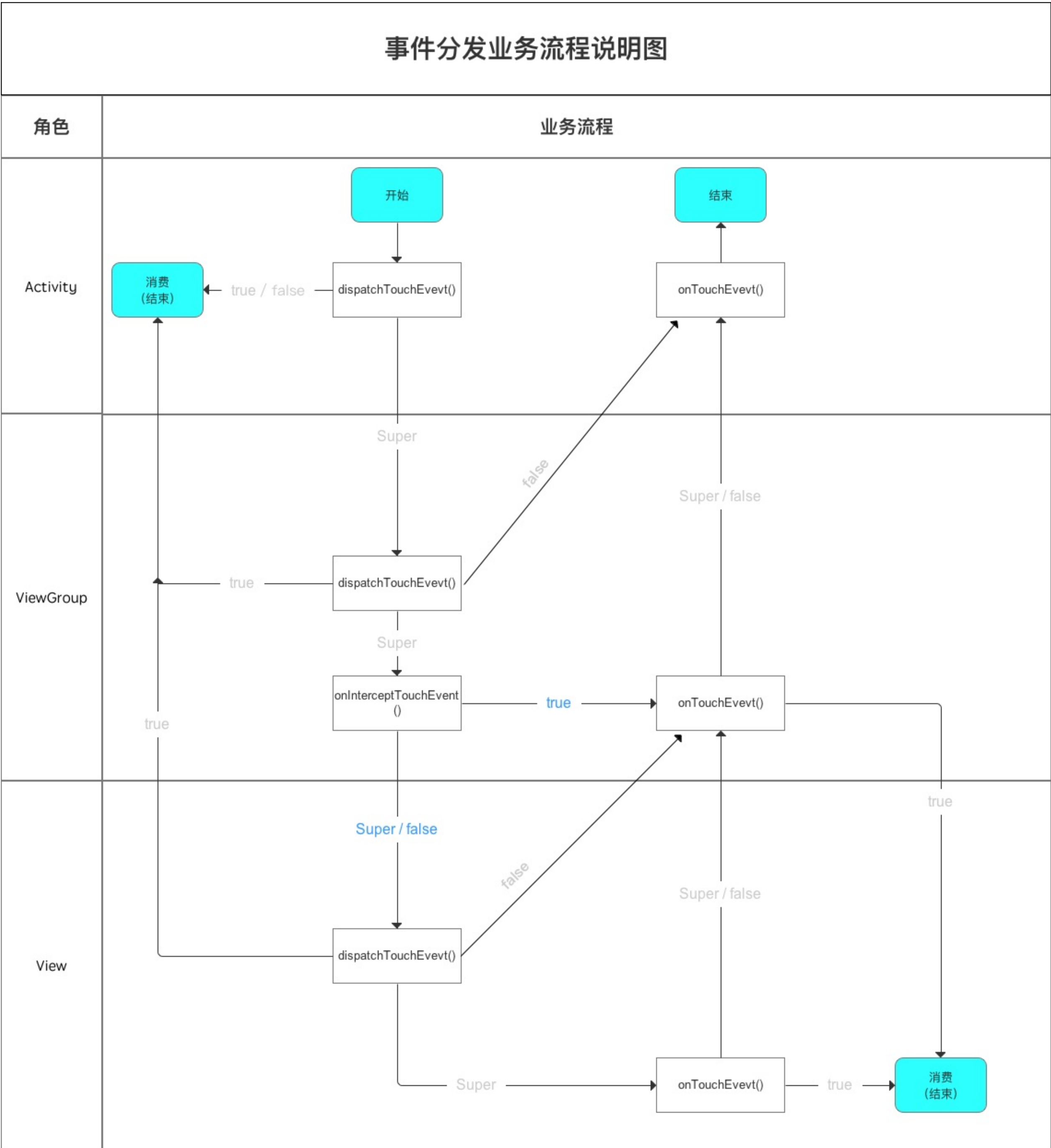
2.3 onInterceptTouchEvent()

属性	介绍
使用对象	ViewGroup（注：Activity、View都没该方法）
作用	拦截事件，即自己处理该事件
调用时刻	在ViewGroup的dispatchTouchEvent()内部调用
返回结果	是否拦截当前事件，详细情况如下：

返回结果

方法	返回结果		
onInterceptTouchEvent()	是否拦截当前事件；	true	• 拦截事件 • 事件不会往下传递； • 自己处理事件，即执行自己的onTouchEvent()； • 同一个事件列的其他事件都直接交由该View处理；在同一个事件列中该方法不会再次被调用；
		false（默认）	• 不拦截事件 • 事件会继续往下传递 • 事件传递到子 view， 调用父类View.dispatchTouchEvent() 方法中去处理 • 当前View仍然接受此事件的其他事件（与onTouchEvent()区别）

- 流程图如下：



2.4 三者关系

下面将用一段伪代码来阐述上述三个方法的关系和点击事件传递规则

```
// 点击事件产生后，会直接调用dispatchTouchEvent（）方法
public boolean dispatchTouchEvent(MotionEvent ev) {

    //代表是否消耗事件
    boolean consume = false;


    if (onInterceptTouchEvent(ev)) {
    //如果onInterceptTouchEvent()返回true则代表当前View拦截了点击事件
    //则该点击事件则会交给当前View进行处理
    //即调用onTouchEvent （）方法去处理点击事件
      consume = onTouchEvent (ev) ;

    } else {
      //如果onInterceptTouchEvent()返回false则代表当前View不拦截点击
事件
      //则该点击事件则会继续传递给它的子元素
      //子元素的dispatchTouchEvent（）就会被调用，重复上述过程
      //直到点击事件被最终处理为止
      consume = child.dispatchTouchEvent (ev) ;
    }

    return consume;
   }
```

2.5 总结

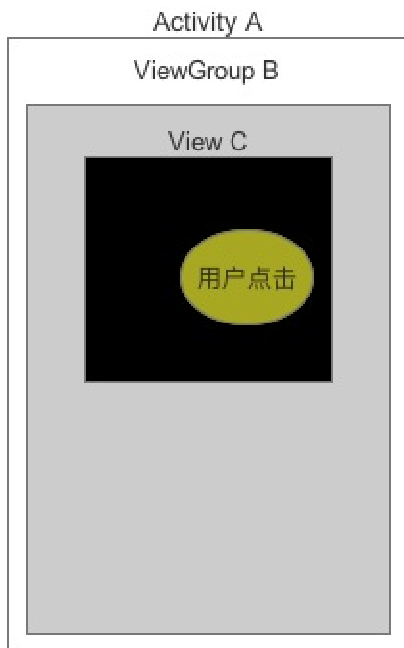
- 对于事件分发的3个方法，你应该清楚了解
- 接下来，我将开始介绍Android事件分发的常见流程

三、事件分发场景介绍

下面我将利用例子来说明常见的点击事件传递情况

3.1 背景描述

我们将要讨论的布局层次如下：



- 最外层：Activiy A，包含两个子View：ViewGroup B、View C
- 中间层：ViewGroup B，包含一个子View：View C
- 最内层：View C

假设用户首先触摸到屏幕上View C上的某个点（如图中黄色区域），那么Action_DOWN事件就在该点产生，然后用户移动手指并最后离开屏幕。

3.2 一般的事件传递情况

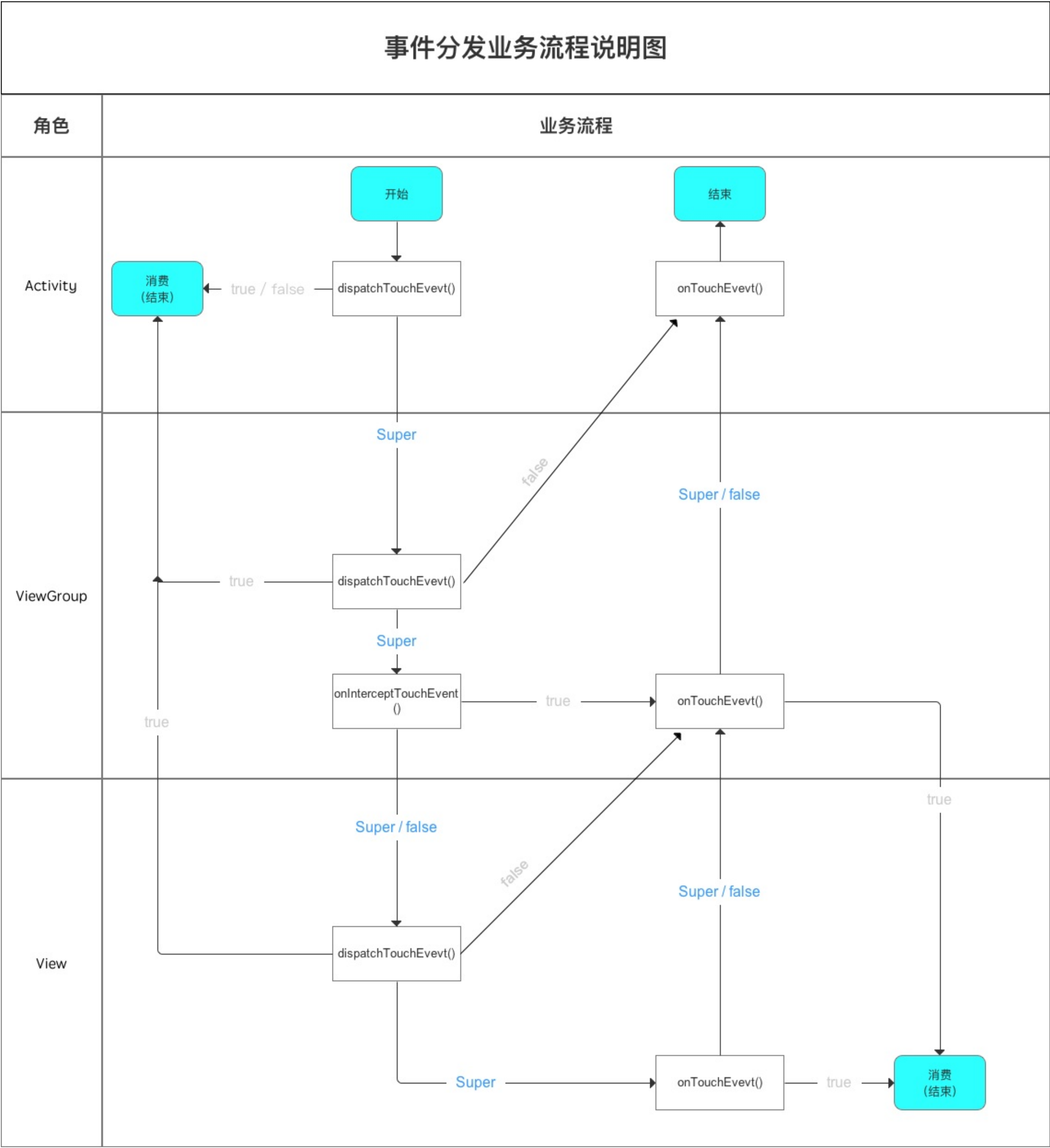
一般的事件传递场景有：

- 默认情况
- 处理事件
- 拦截DOWN事件
- 拦截后续事件（MOVE、UP）

3.2.1 默认情况

- 即不对控件里的方法(dispatchTouchEvent()、onTouchEvent()、onInterceptTouchEvent())进行重写或更改返回值
- 那么调用的是这3个方法的默认实现：调用父类的方法
- 事件传递情况：（如图下所示）
    - 从Activity A---->ViewGroup B--->View C，从上往下调用dispatchTouchEvent()

- 再由View C--->ViewGroup B --->Activity A，从下往上调用onTouchEvent()



注：虽然ViewGroup B的onInterceptTouchEvent方法对DOWN事件返回了false，后续的事件（MOVE、UP）依然会传递给它的onInterceptTouchEvent()

> 这一点与onTouchEvent的行为是不一样的。

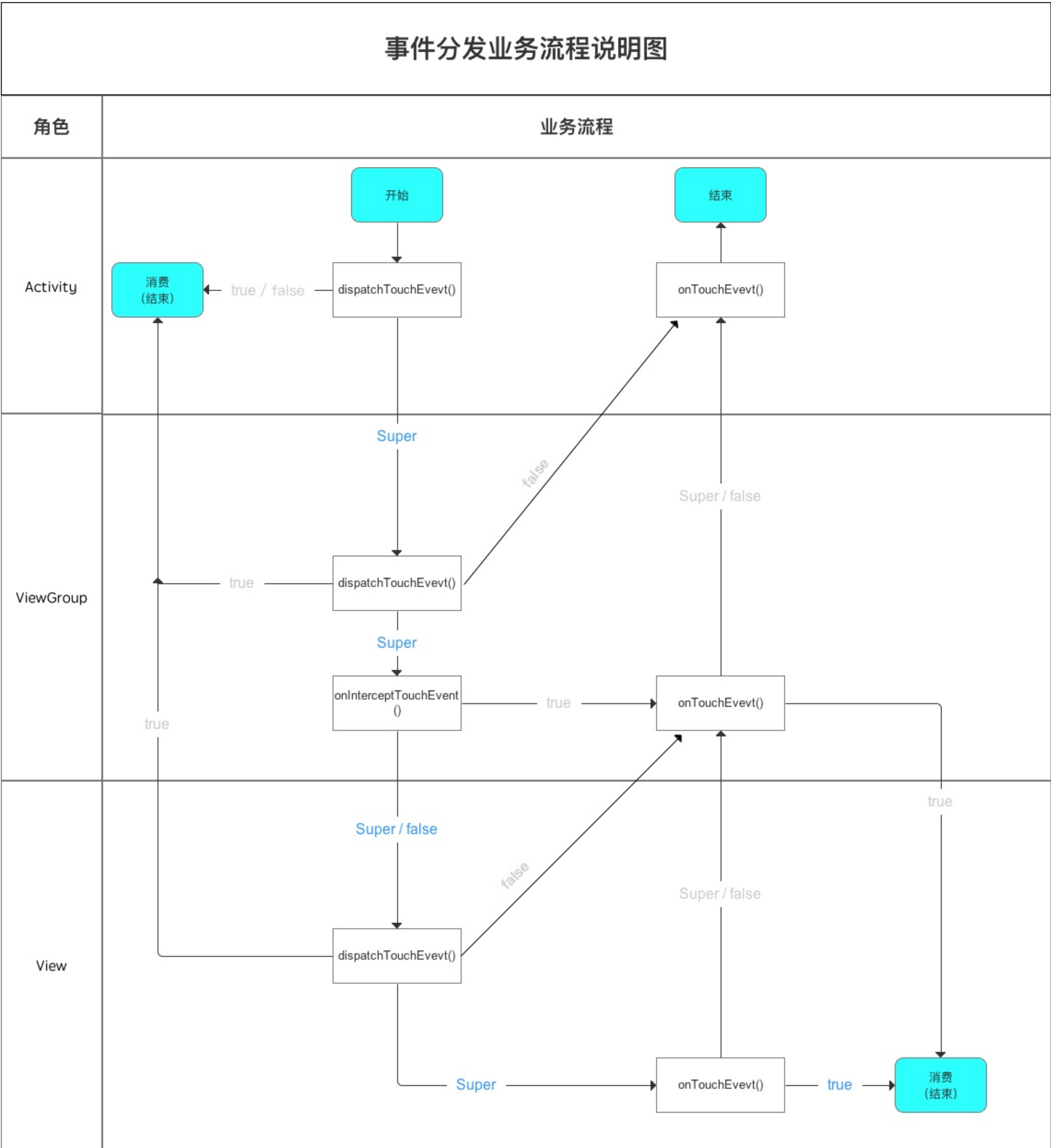
3.2.2 处理事件

假设View C希望处理这个点击事件，即C被设置成可点击的（Clickable）或者覆写了C的onTouchEvent方法返回true。

> 最常见的：设置Button按钮来响应点击事件

事件传递情况：（如下图）

- DOWN事件被传递给C的onTouchEvent方法，该方法返回true，表示处理这个事件
- 因为C正在处理这个事件，那么DOWN事件将不再往上传递给B和A的onTouchEvent()；
- 该事件列的其他事件（Move、Up）也将传递给C的onTouchEvent()



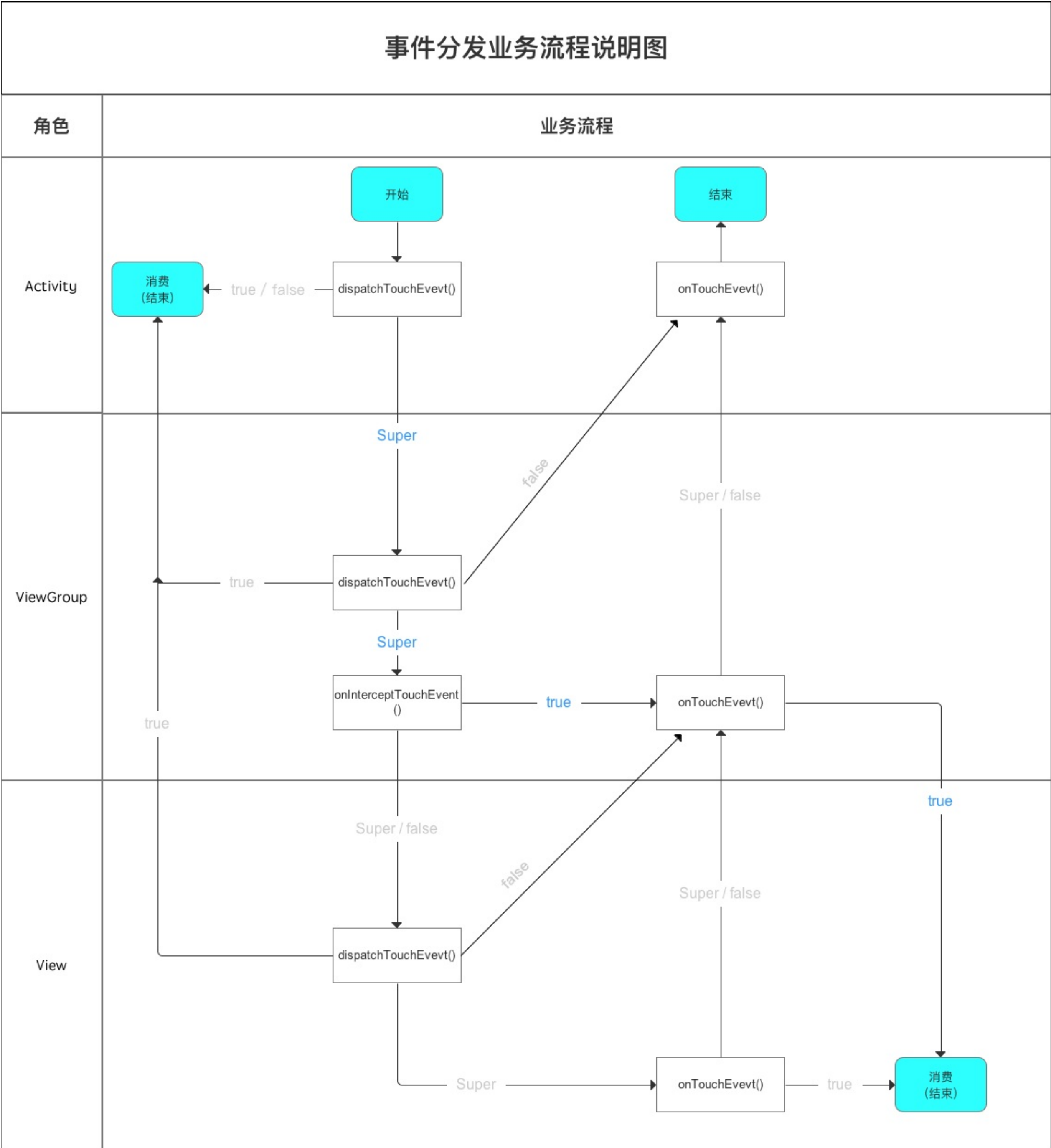
3.2.3 拦截DOWN事件

假设ViewGroup B希望处理这个点击事件，即B覆写了onInterceptTouchEvent()返回true、onTouchEvent()返回true。 事件传递情况：（如下图）

- DOWN事件被传递给B的onInterceptTouchEvent()方法，该方法返回true，表示拦截这个事件，即自己处理这个事件（不再往下传递）
- 调用onTouchEvent()处理事件（DOWN事件将不再往上传递给A的onTouchEvent()）

- 该事件列的其他事件（Move、Up）将直接传递给B的onTouchEvent()

> 该事件列的其他事件（Move、Up）将不会再传递给B的onInterceptTouchEvent方法，该方法一旦返回一次true，就再也不会被调用了。



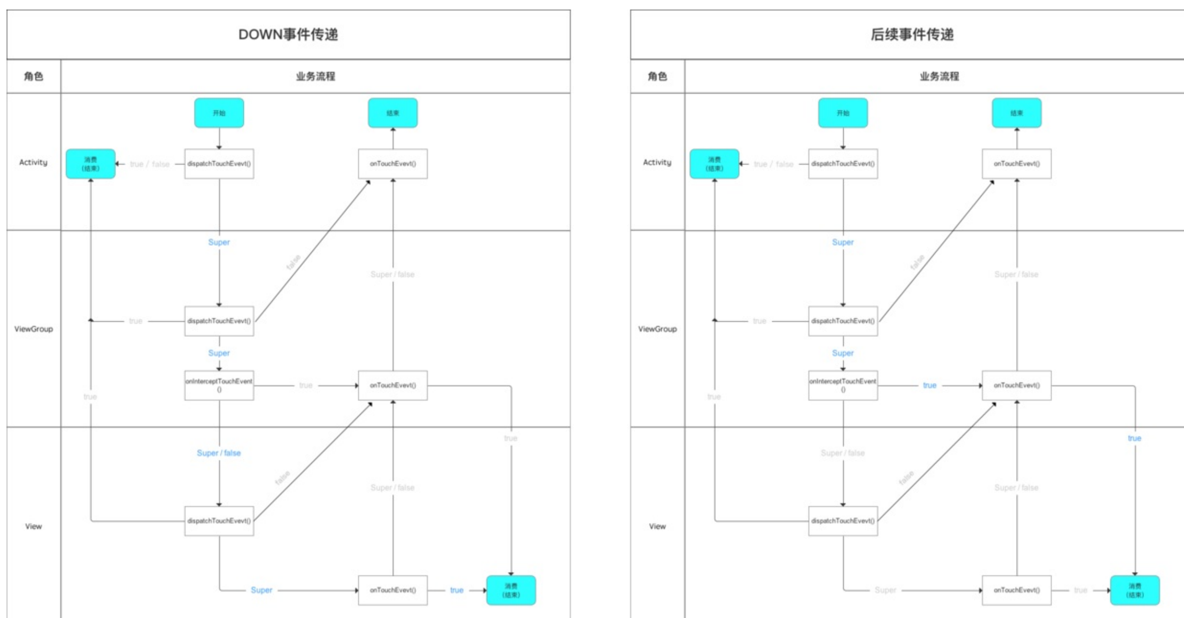
3.2.4 拦截DOWN的后续事件

假设ViewGroup B没有拦截DOWN事件（还是View C来处理DOWN事件），但它拦截了接下来的MOVE事件。

- DOWN事件传递到C的onTouchEvent方法，返回了true。
- 在后续到来的MOVE事件，B的onInterceptTouchEvent方法返回true拦截该MOVE事件，但该事件并没有传递给B；这个MOVE事件将会被系统变成一个CANCEL事件传递给C的onTouchEvent方法
- 后续又来了一个MOVE事件，该MOVE事件才会直接传递给B的onTouchEvent()

> 1. 后续事件将直接传递给B的onTouchEvent()处理
> 2. 后续事件将不会再传递给B的onInterceptTouchEvent方法，该方法一旦返回一次true，就再也不会被调用了。

- C再也不会收到该事件列产生的后续事件。



特别注意：

- 如果ViewGroup A 拦截了一个半路的事件（如MOVE），这个事件将会被系统变成一个CANCEL事件并传递给之前处理该事件的子View；
- 该事件不会再传递给ViewGroup A的onTouchEvent()
- 只有再到来的事件才会传递到ViewGroup A的onTouchEvent()

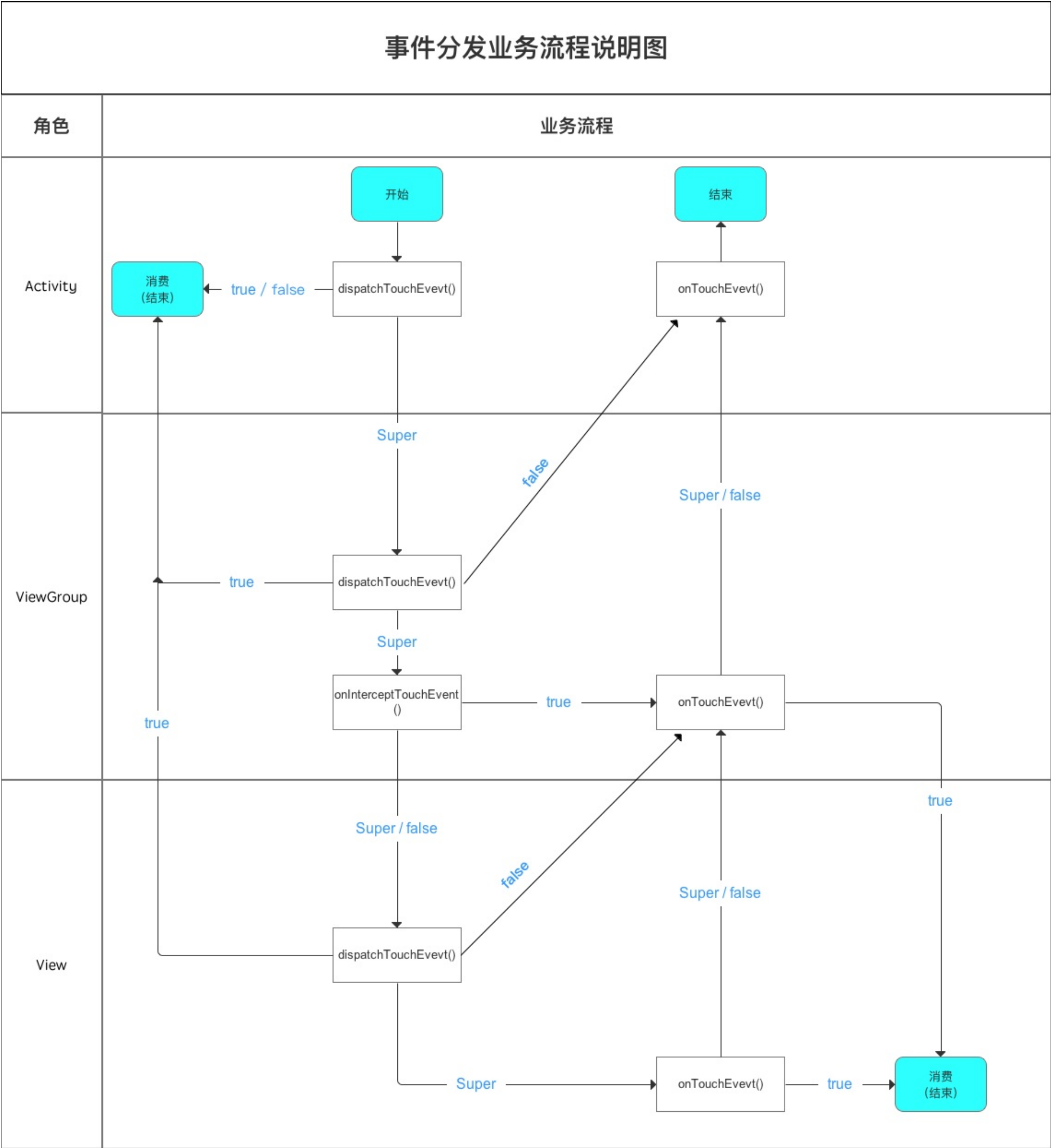
3.3 总结

- 对于Android的事件分发机制，你应该已经非常清楚了
- 如果你只是希望了解**Android**事件分发机制而不想深入了解，那么你可以离开这篇文章了

- 对于程序猿来说，知其然还需要知其所以然，接下来，我将通过源码分析来深入了解**Android**事件分发机制

四、Android事件分发机制源码分析

- Android中事件分发顺序：**Activity**（**Window**）**-> ViewGroup -> View**，再次贴出下图：



其中：

- super：调用父类方法
- true：消费事件，即事件不继续往下传递
- false：不消费事件，事件继续往下传递 / 交由给父控件onTouchEvent（）处理

所以，要想充分理解**Android**分发机制，本质上是要理解：

- Activity对点击事件的分发机制
- ViewGroup对点击事件的分发机制
- View对点击事件的分发机制

接下来，我将通过源码分析详细介绍Activity、View和ViewGroup的事件分发机制

4.1 Activity的事件分发机制

4.1.1 源码分析

- 当一个点击事件发生时，事件最先传到Activity的dispatchTouchEvent()进行事件分发

  > 具体是由Activity的Window来完成

- 我们来看下Activity的dispatchTouchEvent()的源码

```
public boolean dispatchTouchEvent(MotionEvent ev) {
        //关注点1
        //一般事件列开始都是DOWN，所以这里基本是true
        if (ev.getAction() == MotionEvent.ACTION_DOWN) {
            //关注点2
            onUserInteraction();
        }
        //关注点3
        if (getWindow().superDispatchTouchEvent(ev)) {
            return true;
        }
        return onTouchEvent(ev);
    }
```

关注点**1** 一般事件列开始都是DOWN（按下按钮），所以这里返回true，执行onUserInteraction()

关注点**2** 先来看下onUserInteraction()源码

```
  /**
     * Called whenever a key, touch, or trackball event is dispa
tched to the
     * activity.  Implement this method if you wish to know that
 the user has
     * interacted with the device in some way while your activit
y is running.
     * This callback and {@link #onUserLeaveHint} are intended t
o help
     * activities manage status bar notifications intelligently;
 specifically,
     * for helping activities determine the proper time to cance
l a notfication.
     *
     * <p>All calls to your activity's {@link #onUserLeaveHint}
callback will
     * be accompanied by calls to {@link #onUserInteraction}.  T
his
     * ensures that your activity will be told of relevant user
activity such
     * as pulling down the notification pane and touching an ite
m there.
     *
     * <p>Note that this callback will be invoked for the touch
down action
     * that begins a touch gesture, but may not be invoked for t
he touch-moved
     * and touch-up actions that follow.
     *
     * @see #onUserLeaveHint()
     */
public void onUserInteraction() {
}
```

从源码可以看出：

- 该方法为空方法

- 从注释得知：当此activity在栈顶时，触屏点击按home，back，menu键等都会触发此方法
- 所以onUserInteraction()主要用于屏保

关注点3

- Window类是抽象类，且PhoneWindow是Window类的唯一实现类
- superDispatchTouchEvent(ev)是抽象方法
- 通过PhoneWindow类中看一下superDispatchTouchEvent()的作用

```
@Override
public boolean superDispatchTouchEvent(MotionEvent event) {
  return mDecor.superDispatchTouchEvent(event);
//mDecor是DecorView的实例
//DecorView是视图的顶层view，继承自FrameLayout，是所有界面的父类
}
```

- 接下来我们看mDecor.superDispatchTouchEvent(event)：

```
public boolean superDispatchTouchEvent(MotionEvent event) {
    return super.dispatchTouchEvent(event);
//DecorView继承自FrameLayout
//那么它的父类就是ViewGroup
而super.dispatchTouchEvent(event)方法，其实就应该是ViewGroup的dispatchTouchEvent()

}
```

所以：

- 执行**getWindow().superDispatchTouchEvent(ev)**实际上是执行了**ViewGroup.dispatchTouchEvent(event)**
- 再回到最初的代码：

```
public boolean dispatchTouchEvent(MotionEvent ev) {

        if (ev.getAction() == MotionEvent.ACTION_DOWN) {
            //关注点2
            onUserInteraction();
        }
        //关注点3
        if (getWindow().superDispatchTouchEvent(ev)) {
            return true;
        }
        return onTouchEvent(ev);
    }
```

由于一般事件列开始都是DOWN，所以这里返回true，基本上都会进入 `getWindow().superDispatchTouchEvent(ev)` 的判断

- 所以，执行**Activity.dispatchTouchEvent(ev)**实际上是执行了**ViewGroup.dispatchTouchEvent(event)**
- 这样事件就从 Activity 传递到了 ViewGroup

4.1.2 汇总

当一个点击事件发生时，调用顺序如下

1. 事件最先传到Activity的dispatchTouchEvent()进行事件分发
2. 调用Window类实现类PhoneWindow的superDispatchTouchEvent()
3. 调用DecorView的superDispatchTouchEvent()
4. 最终调用DecorView父类的dispatchTouchEvent()，即**ViewGroup**的**dispatchTouchEvent()**

4.1.3 结论

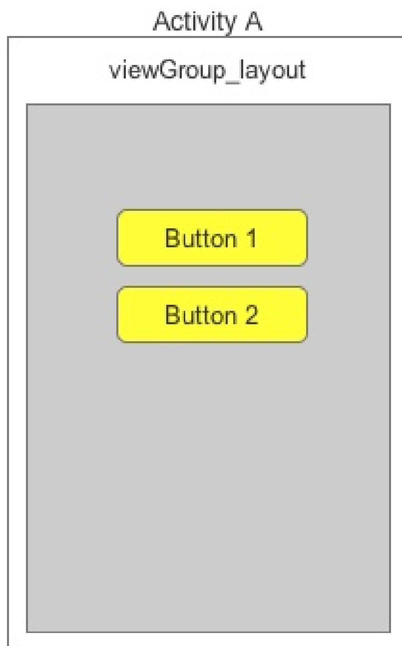
- 当一个点击事件发生时，事件最先传到Activity的dispatchTouchEvent()进行事件分发，最终是调用了ViewGroup的dispatchTouchEvent()方法
- 这样事件就从 Activity 传递到了 ViewGroup

4.2 ViewGroup事件的分发机制

在讲解ViewGroup事件的分发机制之前我们先来看个Demo

4.2.1 Demo讲解

布局如下：



结果测试：只点击Button

```
11-23 22:07:56.929 6149-6149/scut.carson_ho.viewgroup_test I/System.out: 点击了Button1
11-23 22:07:58.185 6149-6149/scut.carson_ho.viewgroup_test I/System.out: 点击了Button2
```

再点击空白处

```
11-23 22:07:56.929 6149-6149/scut.carson_ho.viewgroup_test I/System.out: 点击了Button1
11-23 22:07:58.185 6149-6149/scut.carson_ho.viewgroup_test I/System.out: 点击了Button2
11-23 22:09:05.197 6149-6149/scut.carson_ho.viewgroup_test I/System.out: 点击了ViewGroup
```

从上面的测试结果发现：

- 当点击Button时，执行Button的onClick()，但ViewGroupLayout注册的onTouch（）不会执行
- 只有点击空白区域时才会执行ViewGroupLayout的onTouch（）；
- 结论：Button的onClick()将事件消费掉了，因此事件不会再继续向下传递。

接下来，我们开始进行ViewGroup事件分发的源码分析

4.2.2 源码分析

ViewGroup的dispatchTouchEvent()源码分析,该方法比较复杂，篇幅有限，就截取几个重要的逻辑片段进行介绍，来解析整个分发流程。

```
// 发生ACTION_DOWN事件或者已经发生过ACTION_DOWN,并且将mFirstTouchTarget赋值，才进入此区域，主要功能是拦截器
        final boolean intercepted;
        if (actionMasked == MotionEvent.ACTION_DOWN|| mFirstTouchTarget != null) {
            //disallowIntercept：是否禁用事件拦截的功能(默认是false),即不禁用
            //可以在子View通过调用requestDisallowInterceptTouchEvent方法对这个值进行修改，不让该View拦截事件
            final boolean disallowIntercept = (mGroupFlags & FLAG_DISALLOW_INTERCEPT) != 0;
            //默认情况下会进入该方法
            if (!disallowIntercept) {
                //调用拦截方法
                intercepted = onInterceptTouchEvent(ev);
                ev.setAction(action);
            } else {
                intercepted = false;
            }
        } else {
            // 当没有触摸targets，且不是down事件时，开始持续拦截触摸。
            intercepted = true;
        }
```

这一段的内容主要是为判断是否拦截。如果当前事件的MotionEvent.ACTION_DOWN，则进入判断，调用ViewGroup onInterceptTouchEvent()方法的值，判断是否拦截。如果mFirstTouchTarget != null，即已经发生过MotionEvent.ACTION_DOWN，并且该事件已经有ViewGroup的子View进行处理了，那么也进入判断，调用ViewGroup onInterceptTouchEvent()方法的值，判断是否拦截。如果不是以上两种情况，即已经是MOVE或UP事件了，并且之前的事件没有对象进行处理，则设置成true，开始拦截接下来的所有事件。这也就解释了如果子**View**的**onTouchEvent()**方法返回**false**，那么接下来的一些列事件都不会交给他处理。如果**VieGroup**的**onInterceptTouchEvent()**第一次执行为**true**，则**mFirstTouchTarget = null**，则也会使得接下来不会调用**onInterceptTouchEvent()**，直接将拦截设置为**true**。

当ViewGroup不拦截事件的时候，事件会向下分发交由它的子View或ViewGroup进行处理。

```
  /* 从最底层的父视图开始遍历，
   ** 找寻newTouchTarget，即上面的mFirstTouchTarget
   ** 如果已经存在找寻newTouchTarget，说明正在接收触摸事件，则跳出循环。
    */
for (int i = childrenCount - 1; i >= 0; i--) {
  final int childIndex = customOrder
    ? getChildDrawingOrder(childrenCount, i) : i;
  final View child = (preorderedList == null)
    ? children[childIndex] : preorderedList.get(childIndex);

  // 如果当前视图无法获取用户焦点，则跳过本次循环
  if (childWithAccessibilityFocus != null) {
     if (childWithAccessibilityFocus != child) {
        continue;
     }
     childWithAccessibilityFocus = null;
     i = childrenCount - 1;
  }
  //如果view不可见，或者触摸的坐标点不在view的范围内，则跳过本次循环
  if (!canViewReceivePointerEvents(child)
      || !isTransformedTouchPointInView(x, y, child, null)) {
    ev.setTargetAccessibilityFocus(false);
    continue;
    }

   newTouchTarget = getTouchTarget(child);
   // 已经开始接收触摸事件,并退出整个循环。
   if (newTouchTarget != null) {
       newTouchTarget.pointerIdBits |= idBitsToAssign;
       break;
    }

    //重置取消或抬起标志位
    //如果触摸位置在child的区域内，则把事件分发给子View或ViewGroup
    if (dispatchTransformedTouchEvent(ev, false, child, idBitsTo
Assign)) {
        // 获取TouchDown的时间点
```

```
            mLastTouchDownTime = ev.getDownTime();
            // 获取TouchDown的Index
            if (preorderedList != null) {
               for (int j = 0; j < childrenCount; j++) {
                   if (children[childIndex] == mChildren[j]) {
                        mLastTouchDownIndex = j;
                        break;
                    }
               }
             } else {
                     mLastTouchDownIndex = childIndex;
                    }

          //获取TouchDown的x,y坐标
          mLastTouchDownX = ev.getX();
          mLastTouchDownY = ev.getY();
          //添加TouchTarget,则mFirstTouchTarget != null。
          newTouchTarget = addTouchTarget(child, idBitsToAssign);
          //表示以及分发给NewTouchTarget
          alreadyDispatchedToNewTouchTarget = true;
          break;
  }
```

`dispatchTransformedTouchEvent()` 方法实际就是调用子元素的 `dispatchTouchEvent()` 方法。其中 `dispatchTransformedTouchEvent()` 方法的重要逻辑如下：

```
  if (child == null) {
                handled = super.dispatchTouchEvent(event);
            } else {
                handled = child.dispatchTouchEvent(event);
            }
```

由于其中传递的child不为空，所以就会调用子元素的dispatchTouchEvent()。如果子元素的dispatchTouchEvent()方法返回true，那么mFirstTouchTarget就会被赋值，同时跳出for循环。

```
//添加TouchTarget,则mFirstTouchTarget != null。
newTouchTarget = addTouchTarget(child, idBitsToAssign);
 //表示以及分发给NewTouchTarget
 alreadyDispatchedToNewTouchTarget = true;
```

其中在 `addTouchTarget(child, idBitsToAssign);` 内部完成mFirstTouchTarget被赋值。 如果mFirstTouchTarget为空，将会让ViewGroup默认拦截所有操作。 如果遍历所有子View或ViewGroup，都没有消费事件。ViewGroup会自己处理事件。

结论

- Android事件分发是先传递到ViewGroup，再由ViewGroup传递到View
- 在ViewGroup中通过onInterceptTouchEvent()对事件传递进行拦截

> 1. onInterceptTouchEvent方法返回true代表拦截事件，即不允许事件继续向子View传递；
> 2. 返回false代表不拦截事件，即允许事件继续向子View传递；（默认返回false）
> 3. 子View中如果将传递的事件消费掉，ViewGroup中将无法接收到任何事件。

4.3 View事件的分发机制

View中dispatchTouchEvent()的源码分析

```
public boolean dispatchTouchEvent(MotionEvent event) {
    if (mOnTouchListener != null && (mViewFlags & ENABLED_MASK)
== ENABLED &&
            mOnTouchListener.onTouch(this, event)) {
        return true;
    }
    return onTouchEvent(event);
}
```

从上面可以看出：

- 只有以下三个条件都为真，dispatchTouchEvent()才返回true；否则执行onTouchEvent(event)方法

```
第一个条件：mOnTouchListener != null；
第二个条件：(mViewFlags & ENABLED_MASK) == ENABLED；
第三个条件：mOnTouchListener.onTouch(this, event)；
```

- 下面，我们来看看下这三个判断条件：

第一个条件：**mOnTouchListener!= null**

```
//mOnTouchListener是在View类下setOnTouchListener方法里赋值的
public void setOnTouchListener(OnTouchListener l) {

//即只要我们给控件注册了Touch事件，mOnTouchListener就一定被赋值（不为空）

    mOnTouchListener = l;
}
```

第二个条件：**(mViewFlags & ENABLED_MASK) == ENABLED**

- 该条件是判断当前点击的控件是否enable
- 由于很多View默认是enable的，因此该条件恒定为true

第三个条件：**mOnTouchListener.onTouch(this, event)**

- 回调控件注册Touch事件时的onTouch方法

```
//手动调用设置
button.setOnTouchListener(new OnTouchListener() {

  @Override
  public boolean onTouch(View v, MotionEvent event) {

      return false;
  }
});
```

- 如果在onTouch方法返回true，就会让上述三个条件全部成立，从而整个方法直接返回true。
- 如果在onTouch方法里返回false，就会去执行onTouchEvent(event)方法。

接下来，我们继续看：**onTouchEvent(event)**的源码分析

```
public boolean onTouchEvent(MotionEvent event) {
    final int viewFlags = mViewFlags;
    if ((viewFlags & ENABLED_MASK) == DISABLED) {
        // A disabled view that is clickable still consumes the
touch
        // events, it just doesn't respond to them.
        return (((viewFlags & CLICKABLE) == CLICKABLE ||
                (viewFlags & LONG_CLICKABLE) == LONG_CLICKABLE))
;
    }
    if (mTouchDelegate != null) {
        if (mTouchDelegate.onTouchEvent(event)) {
            return true;
        }
    }
     //如果该控件是可以点击的就会进入到下两行的switch判断中去；

    if (((viewFlags & CLICKABLE) == CLICKABLE ||
            (viewFlags & LONG_CLICKABLE) == LONG_CLICKABLE)) {
    //如果当前的事件是抬起手指，则会进入到MotionEvent.ACTION_UP这个case
当中。

        switch (event.getAction()) {
            case MotionEvent.ACTION_UP:
                boolean prepressed = (mPrivateFlags & PREPRESSED
) != 0;
               // 在经过种种判断之后，会执行到关注点1的performClick()
方法。
               //请往下看关注点1
                if ((mPrivateFlags & PRESSED) != 0 || prepressed
) {
                    // take focus if we don't have it already an
d we should in
```

```
                    // touch mode.
                    boolean focusTaken = false;
                    if (isFocusable() && isFocusableInTouchMode(
) && !isFocused()) {
                        focusTaken = requestFocus();
                    }
                    if (!mHasPerformedLongPress) {
                        // This is a tap, so remove the longpres
s check
                        removeLongPressCallback();
                        // Only perform take click actions if we
 were in the pressed state
                        if (!focusTaken) {
                            // Use a Runnable and post this rath
er than calling
                            // performClick directly. This lets
other visual state
                            // of the view update before click a
ctions start.
                            if (mPerformClick == null) {
                                mPerformClick = new PerformClick
();
                            }
                            if (!post(mPerformClick)) {
            //关注点1
            //请往下看performClick()的源码分析
                                performClick();
                            }
                        }
                    }
                    if (mUnsetPressedState == null) {
                        mUnsetPressedState = new UnsetPressedSta
te();
                    }
                    if (prepressed) {
                        mPrivateFlags |= PRESSED;
                        refreshDrawableState();
                        postDelayed(mUnsetPressedState,
                                ViewConfiguration.getPressedStat
eDuration());
```

```
                    } else if (!post(mUnsetPressedState)) {
                        // If the post failed, unpress right now

                        mUnsetPressedState.run();
                    }
                    removeTapCallback();
                }
                break;
            case MotionEvent.ACTION_DOWN:
                if (mPendingCheckForTap == null) {
                    mPendingCheckForTap = new CheckForTap();
                }
                mPrivateFlags |= PREPRESSED;
                mHasPerformedLongPress = false;
                postDelayed(mPendingCheckForTap, ViewConfigurati
on.getTapTimeout());
                break;
            case MotionEvent.ACTION_CANCEL:
                mPrivateFlags &= ~PRESSED;
                refreshDrawableState();
                removeTapCallback();
                break;
            case MotionEvent.ACTION_MOVE:
                final int x = (int) event.getX();
                final int y = (int) event.getY();
                // Be lenient about moving outside of buttons
                int slop = mTouchSlop;
                if ((x < 0 - slop) || (x >= getWidth() + slop) |
|
                        (y < 0 - slop) || (y >= getHeight() + sl
op)) {
                    // Outside button
                    removeTapCallback();
                    if ((mPrivateFlags & PRESSED) != 0) {
                        // Remove any future long press/tap chec
ks

                        removeLongPressCallback();
                        // Need to switch from pressed to not pr
essed

                        mPrivateFlags &= ~PRESSED;
```

```
                            refreshDrawableState();
                        }
                    }
                    break;
            }
//如果该控件是可以点击的，就一定会返回true
            return true;
        }
//如果该控件是不可以点击的，就一定会返回false
        return false;
}
```

关注点**1**： performClick()的源码分析

```
public boolean performClick() {
    sendAccessibilityEvent(AccessibilityEvent.TYPE_VIEW_CLICKED)
;

    if (mOnClickListener != null) {
        playSoundEffect(SoundEffectConstants.CLICK);
        mOnClickListener.onClick(this);
        return true;
    }
    return false;
}
```

- 只要mOnClickListener不为null，就会去调用onClick方法；
- 那么，mOnClickListener又是在哪里赋值的呢？请继续看：

```
public void setOnClickListener(OnClickListener l) {
    if (!isClickable()) {
        setClickable(true);
    }
    mOnClickListener = l;
}
```

- 当我们通过调用setOnClickListener方法来给控件注册一个点击事件时，就会给mOnClickListener赋值（不为空），即会回调onClick（）。

结论

1. onTouch（）的执行高于onClick（）
2. 每当控件被点击时：
    - 如果在回调onTouch()里返回false，就会让dispatchTouchEvent方法返回false，那么就会执行onTouchEvent()；如果回调了setOnClickListener()来给控件注册点击事件的话，最后会在performClick()方法里回调onClick()。

      > onTouch()返回false（该事件没被onTouch()消费掉） = 执行onTouchEvent() = 执行OnClick()

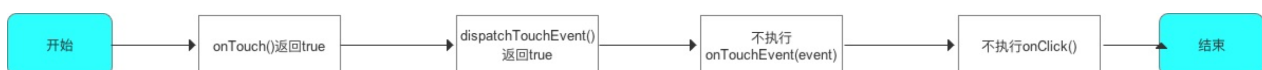
    - 如果在回调onTouch()里返回true，就会让dispatchTouchEvent方法返回true，那么将不会执行onTouchEvent()，即onClick()也不会执行；

      > onTouch()返回true（该事件被onTouch()消费掉） = dispatchTouchEvent()返回true（不会再继续向下传递） = 不会执行onTouchEvent() = 不会执行OnClick()

下面我将用**Demo**验证上述的结论

Demo论证

1. Demo1：在回调onTouch()里返回true



```
//设置OnTouchListener()
    button.setOnTouchListener(new View.OnTouchListener() {

            @Override
            public boolean onTouch(View v, MotionEvent event) {
                System.out.println("执行了onTouch(), 动作是:" + event.getAction());

                return true;
            }
        });

//设置OnClickListener
     button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                System.out.println("执行了onClick()");
            }
        });
```

点击Button，测试结果如下：

```
11-23 04:15:54.578 7334-7334/scut.carson_ho.touch_event I/System.out: 执行了onTouch(), 动作是:0
11-23 04:15:54.638 7334-7334/scut.carson_ho.touch_event I/System.out: 执行了onTouch(), 动作是:1
```

2. Demo2：在回调onTouch()里返回false

```
//设置OnTouchListener()
   button.setOnTouchListener(new View.OnTouchListener() {

            @Override
            public boolean onTouch(View v, MotionEvent event) {
                System.out.println("执行了onTouch(), 动作是:" + ev
ent.getAction());

                return false;
            }
        });

//设置OnClickListener
    button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                System.out.println("执行了onClick()");
            }
        });
```

点击Button，测试结果如下：

```
11-23 04:08:36.514 1703-1703/scut.carson_ho.touch_event I/System.out: 执行了onTouch(), 动作是:0
11-23 04:08:36.578 1703-1703/scut.carson_ho.touch_event I/System.out: 执行了onTouch(), 动作是:1
11-23 04:08:36.578 1703-1703/scut.carson_ho.touch_event I/System.out: 执行了onClick()
```

总结：**onTouch()**返回**true**就认为该事件被**onTouch()**消费掉，因而不会再继续向下传递，即不会执行**OnClick()**。

五、 思考点

5.1 onTouch()和onTouchEvent()的区别

- 这两个方法都是在View的dispatchTouchEvent中调用，但onTouch优先于onTouchEvent执行。
- 如果在onTouch方法中返回true将事件消费掉，onTouchEvent()将不会再执行。

- 特别注意：请看下面代码

```
//&&为短路与，即如果前面条件为false，将不再往下执行
//所以，onTouch能够得到执行需要两个前提条件：
//1. mOnTouchListener的值不能为空
//2. 当前点击的控件必须是enable的。
mOnTouchListener != null && (mViewFlags & ENABLED_MASK) == E
NABLED &&
          mOnTouchListener.onTouch(this, event)
```

- 因此如果你有一个控件是非enable的，那么给它注册onTouch事件将永远得不到执行。对于这一类控件，如果我们想要监听它的touch事件，就必须通过在该控件中重写onTouchEvent方法来实现。

5.2 Touch事件的后续事件（MOVE、UP）层级传递

- 如果给控件注册了Touch事件，每次点击都会触发一系列action事件（ACTION_DOWN，ACTION_MOVE，ACTION_UP等）
- 当dispatchTouchEvent在进行事件分发的时候，只有前一个事件（如ACTION_DOWN）返回true，才会收到后一个事件（ACTION_MOVE和ACTION_UP）

> 即如果在执行ACTION_DOWN时返回false，后面一系列的ACTION_MOVE和ACTION_UP事件都不会执行

从上面对事件分发机制分析知：

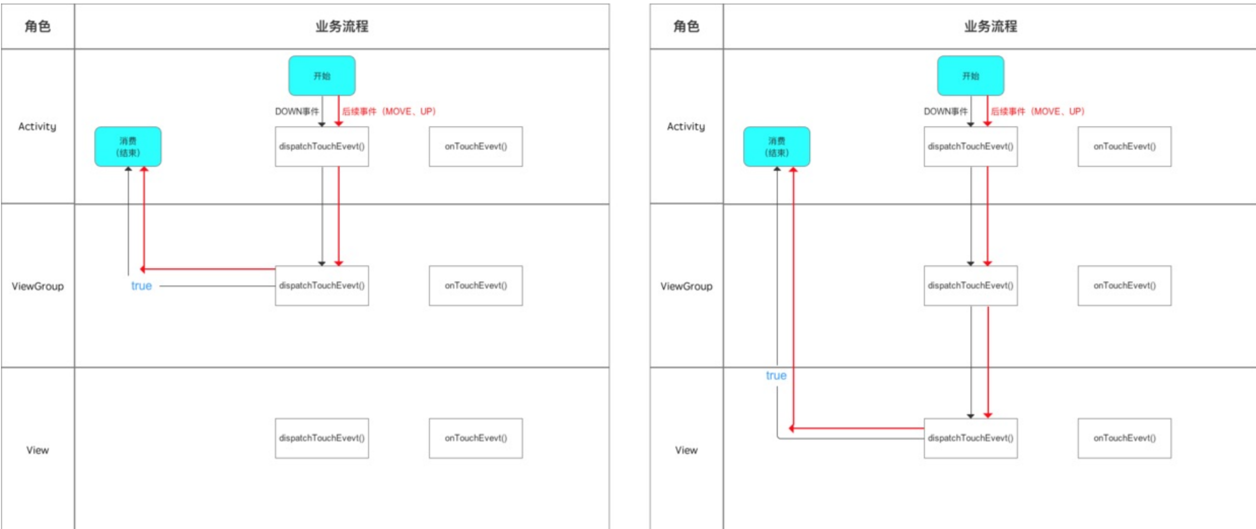
- dispatchTouchEvent()和 onTouchEvent()消费事件、终结事件传递（返回true）
- 而onInterceptTouchEvent 并不能消费事件，它相当于是一个分叉口起到分流导流的作用，对后续的ACTION_MOVE和ACTION_UP事件接收起到非常大的作用

> 请记住：接收了ACTION_DOWN事件的函数不一定能收到后续事件（ACTION_MOVE、ACTION_UP）

这里给出**ACTION_MOVE**和**ACTION_UP**事件的传递结论：

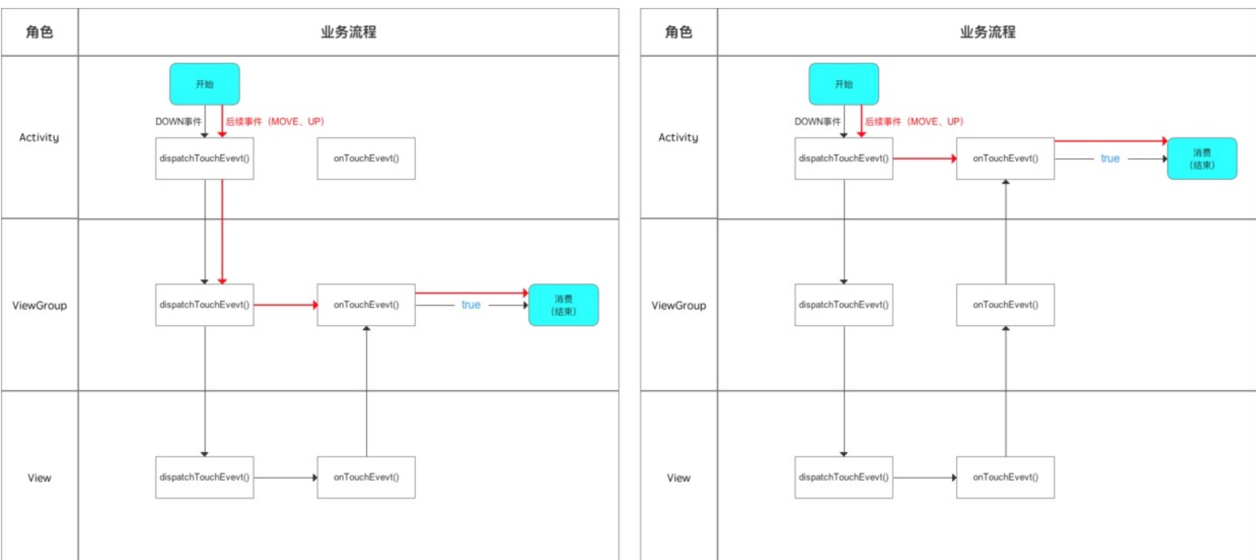
- 如果在某个对象（Activity、ViewGroup、View）的dispatchTouchEvent()消费事件（返回true），那么收到ACTION_DOWN的函数也能收到ACTION_MOVE和ACTION_UP

> 黑线：ACTION_DOWN事件传递方向 红线：ACTION_MOVE和ACTION_UP事件传递方向



- 如果在某个对象（Activity、ViewGroup、View）的onTouchEvent()消费事件（返回true），那么ACTION_MOVE和ACTION_UP的事件从上往下传到这个View后就不再往下传递了，而直接传给自己的onTouchEvent()并结束本次事件传递过程。

> 黑线：ACTION_DOWN事件传递方向 红线：ACTION_MOVE和ACTION_UP事件传递方向



一、Android中的线程

在操作系统中，线程是操作系统调度的最小单元，同时线程又是一种受限的系统资源，即线程不可能无限制地产生，并且线程的创建和销毁都会有相应的开销。当系统中存在大量的线程时，系统会通过会时间片轮转的方式调度每个线程，因此线程不可能做到绝对的并行。

如果在一个进程中频繁地创建和销毁线程，显然不是高效的做法。正确的做法是采用线程池，一个线程池中会缓存一定数量的线程，通过线程池就可以避免因为频繁创建和销毁线程所带来的系统开销。

二、AsyncTask简介

AsyncTask是一个抽象类，它是由Android封装的一个轻量级异步类（轻量体现在使用方便、代码简洁），它可以在线程池中执行后台任务，然后把执行的进度和最终结果传递给主线程并在主线程中更新UI。

AsyncTask的内部封装了两个线程池(SerialExecutor和THREAD_POOL_EXECUTOR)和一个**Handler**(InternalHandler)。

其中**SerialExecutor**线程池用于任务的排队，让需要执行的多个耗时任务，按顺序排列，**THREAD_POOL_EXECUTOR**线程池才真正地执行任务，**InternalHandler**用于从工作线程切换到主线程。

1.AsyncTask的泛型参数

AsyncTask的类声明如下：

```
public abstract class AsyncTask<Params, Progress, Result>
```

AsyncTask是一个抽象泛型类。

其中，三个泛型类型参数的含义如下：

Params：开始异步任务执行时传入的参数类型；

Progress：异步任务执行过程中，返回下载进度值的类型；

Result：异步任务执行完成后，返回的结果类型；

如果**AsyncTask**确定不需要传递具体参数，那么这三个泛型参数可以用**Void**来代替。

有了这三个参数类型之后，也就控制了这个AsyncTask子类各个阶段的返回类型，如果有不同业务，我们就需要再另写一个AsyncTask的子类进行处理。

2.AsyncTask的核心方法

onPreExecute()

这个方法会在后台任务开始执行之间调用，在主线程执行。用于进行一些界面上的初始化操作，比如显示一个进度条对话框等。

doInBackground(Params...)

这个方法中的所有代码都会在子线程中运行，我们应该在这里去处理所有的耗时任务。

任务一旦完成就可以通过return语句来将任务的执行结果进行返回，如果AsyncTask的第三个泛型参数指定的是Void，就可以不返回任务执行结果。注意，在这个方法中是不可以进行**UI**操作的，如果需要更新**UI**元素，比如说反馈当前任务的执行进度，可以调用**publishProgress(Progress...)**方法来完成。

onProgressUpdate(Progress...)

当在后台任务中调用了publishProgress(Progress...)方法后，这个方法就很快会被调用，方法中携带的参数就是在后台任务中传递过来的。在这个方法中可以对**UI**进行操作，在主线程中进行，利用参数中的数值就可以对界面元素进行相应的更新。

onPostExecute(Result)

当doInBackground(Params...)执行完毕并通过return语句进行返回时，这个方法就很快会被调用。返回的数据会作为参数传递到此方法中，可以利用返回的数据来进行一些**UI**操作，在主线程中进行，比如说提醒任务执行的结果，以及关闭掉进度条对话框等。

上面几个方法的调用顺序： onPreExecute() --> doInBackground() --> publishProgress() --> onProgressUpdate() --> onPostExecute()

如果不需要执行更新进度则为onPreExecute() --> doInBackground() --> onPostExecute(),

除了上面四个方法，AsyncTask还提供了onCancelled()方法，它同样在主线程中执行，当异步任务取消时，**onCancelled()**会被调用，这个时候**onPostExecute()**则不会被调用，但是要注意的是，**AsyncTask**中的**cancel()**方法并不是真正去取消任务，只是设置这个任务为取消状态，我们需要在**doInBackground()**判断终止任务。就好比想要终止一个线程，调用**interrupt()**方法，只是进行标记为中断，需要在线程内部进行标记判断然后中断线程。

3.AsyncTask的简单使用

```
class DownloadTask extends AsyncTask<Void, Integer, Boolean> {

    @Override
    protected void onPreExecute() {
        progressDialog.show();
    }

    @Override
    protected Boolean doInBackground(Void... params) {
        try {
            while (true) {
                int downloadPercent = doDownload();
                publishProgress(downloadPercent);
                if (downloadPercent >= 100) {
                    break;
                }
            }
        } catch (Exception e) {
            return false;
        }
        return true;
    }

    @Override
    protected void onProgressUpdate(Integer... values) {
        progressDialog.setMessage("当前下载进度：" + values[0] + "
%");
```

```
    }

    @Override
    protected void onPostExecute(Boolean result) {
        progressDialog.dismiss();
        if (result) {
            Toast.makeText(context, "下载成功", Toast.LENGTH_SHOR
T).show();
        } else {
            Toast.makeText(context, "下载失败", Toast.LENGTH_SHOR
T).show();
        }
    }
}
```

这里我们模拟了一个下载任务，在doInBackground()方法中去执行具体的下载逻辑，在onProgressUpdate()方法中显示当前的下载进度，在onPostExecute()方法中来提示任务的执行结果。如果想要启动这个任务，只需要简单地调用以下代码即可：

```
new DownloadTask().execute();
```

4.使用AsyncTask的注意事项

①异步任务的实例必须在UI线程中创建，即AsyncTask对象必须在UI线程中创建。

②execute(Params... params)方法必须在UI线程中调用。

③不要手动调用onPreExecute()，doInBackground(Params... params)，onProgressUpdate(Progress... values)，onPostExecute(Result result)这几个方法。

④不能在doInBackground(Params... params)中更改UI组件的信息。

⑤一个任务实例只能执行一次，如果执行第二次将会抛出异常。

三、AsyncTask的源码分析

先从初始化一个AsyncTask时，调用的构造函数开始分析。

```
public AsyncTask() {
        mWorker = new WorkerRunnable<Params, Result>() {
            public Result call() throws Exception {
                mTaskInvoked.set(true);
                Result result = null;
                try {
                    Process.setThreadPriority(Process.THREAD_PRI
ORITY_BACKGROUND);
                    //noinspection unchecked
                    result = doInBackground(mParams);
                    Binder.flushPendingCommands();
                } catch (Throwable tr) {
                    mCancelled.set(true);
                    throw tr;
                } finally {
                    postResult(result);
                }
                return result;
            }
        };

        mFuture = new FutureTask<Result>(mWorker) {
            @Override
            protected void done() {
                try {
                    postResultIfNotInvoked(get());
                } catch (InterruptedException e) {
                    android.util.Log.w(LOG_TAG, e);
                } catch (ExecutionException e) {
                    throw new RuntimeException("An error occurre
d while executing doInBackground()",
                            e.getCause());
                } catch (CancellationException e) {
                    postResultIfNotInvoked(null);
                }
            }
        };
    }
```

这段代码虽然看起来有点长，但实际上并没有任何具体的逻辑会得到执行，只是初始化了两个变量，mWorker和mFuture，并在初始化mFuture的时候将mWorker作为参数传入。mWorker是一个Callable对象，mFuture是一个FutureTask对象，这两个变量会暂时保存在内存中，稍后才会用到它们。 FutureTask实现了Runnable接口。

mWorker中的call()方法执行了耗时操作，即 `result = doInBackground(mParams);` ,然后把执行得到的结果通过 `postResult(result);` ,传递给内部的Handler跳转到主线程中。在这里这是实例化了两个变量，并没有开启执行任务。

那么**mFuture**对象是怎么加载到线程池中，进行执行的呢？

接着如果想要启动某一个任务，就需要调用该任务的execute()方法，因此现在我们来看一看execute()方法的源码，如下所示：

```
 public final AsyncTask<Params, Progress, Result> execute(Params
... params) {
        return executeOnExecutor(sDefaultExecutor, params);
    }
```

调用了executeOnExecutor()方法,具体执行逻辑在这个方法里面：

```
  public final AsyncTask<Params, Progress, Result> executeOnExec
utor(Executor exec,
            Params... params) {
        if (mStatus != Status.PENDING) {
            switch (mStatus) {
                case RUNNING:
                    throw new IllegalStateException("Cannot exec
ute task:"
                            + " the task is already running.");
                case FINISHED:
                    throw new IllegalStateException("Cannot exec
ute task:"
                            + " the task has already been execut
ed "
                            + "(a task can be executed only once
)");
            }
        }

        mStatus = Status.RUNNING;

        onPreExecute();

        mWorker.mParams = params;
        exec.execute(mFuture);

        return this;
    }
```

可以 看出，先执行了onPreExecute()方法，然后具体执行耗时任务是在 `exec.execute(mFuture)` ，把构造函数中实例化的mFuture传递进去了。

exec具体是什么？

从上面可以看出具体是sDefaultExecutor，再追溯看到是SerialExecutor类，具体源码如下：

```
private static class SerialExecutor implements Executor {
        final ArrayDeque<Runnable> mTasks = new ArrayDeque<Runna
ble>();
        Runnable mActive;

        public synchronized void execute(final Runnable r) {
            mTasks.offer(new Runnable() {
                public void run() {
                    try {
                        r.run();
                    } finally {
                        scheduleNext();
                    }
                }
            });
            if (mActive == null) {
                scheduleNext();
            }
        }

        protected synchronized void scheduleNext() {
            if ((mActive = mTasks.poll()) != null) {
                THREAD_POOL_EXECUTOR.execute(mActive);
            }
        }
    }
```

终于追溯到了调用了SerialExecutor 类的execute方法。SerialExecutor 是个静态内部类，是所有实例化的AsyncTask对象公有的，SerialExecutor 内部维持了一个队列，通过锁使得该队列保证AsyncTask中的任务是串行执行的，即多个任务需要一个个加到该队列中，然后执行完队列头部的再执行下一个，以此类推。

在这个方法中，有两个主要步骤。①向队列中加入一个新的任务，即之前实例化后的mFuture对象。

②调用 `scheduleNext()` 方法，调用THREAD_POOL_EXECUTOR执行队列头部的任务。

由此可见**SerialExecutor** 类仅仅为了保持任务执行是串行的，实际执行交给了**THREAD_POOL_EXECUTOR**。

THREAD_POOL_EXECUTOR又是什么？

```
 ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor(
                CORE_POOL_SIZE, MAXIMUM_POOL_SIZE, KEEP_ALIVE_SE
CONDS, TimeUnit.SECONDS,
                sPoolWorkQueue, sThreadFactory);
        threadPoolExecutor.allowCoreThreadTimeOut(true);
        THREAD_POOL_EXECUTOR = threadPoolExecutor;
```

实际是个线程池，开启了一定数量的核心线程和工作线程。然后调用线程池的execute()方法。执行具体的耗时任务，即开头构造函数中mWorker中call()方法的内容。先执行完doInBackground()方法，又执行postResult()方法，下面看该方法的具体内容：

```
 private Result postResult(Result result) {
        @SuppressWarnings("unchecked")
        Message message = getHandler().obtainMessage(MESSAGE_POS
T_RESULT,
                new AsyncTaskResult<Result>(this, result));
        message.sendToTarget();
        return result;
    }
```

该方法向Handler对象发送了一个消息，下面具体看AsyncTask中实例化的Hanlder对象的源码：

```
private static class InternalHandler extends Handler {
        public InternalHandler() {
            super(Looper.getMainLooper());
        }

        @SuppressWarnings({"unchecked", "RawUseOfParameterizedTy
pe"})
        @Override
        public void handleMessage(Message msg) {
            AsyncTaskResult<?> result = (AsyncTaskResult<?>) msg
.obj;
            switch (msg.what) {
                case MESSAGE_POST_RESULT:
                    // There is only one result
                    result.mTask.finish(result.mData[0]);
                    break;
                case MESSAGE_POST_PROGRESS:
                    result.mTask.onProgressUpdate(result.mData);
                    break;
            }
        }
    }
```

在InternalHandler 中，如果收到的消息是MESSAGE_POST_RESULT，即执行完了doInBackground()方法并传递结果，那么就调用finish()方法。

```
private void finish(Result result) {
        if (isCancelled()) {
            onCancelled(result);
        } else {
            onPostExecute(result);
        }
        mStatus = Status.FINISHED;
    }
```

如果任务已经取消了，回调onCancelled()方法，否则回调 onPostExecute()方法。

如果收到的消息是MESSAGE_POST_PROGRESS，回调onProgressUpdate()方法，更新进度。

InternalHandler是一个静态类，为了能够将执行环境切换到主线程，因此这个类必须在主线程中进行加载。所以变相要求**AsyncTask**的类必须在主线程中进行加载。

到此为止，从任务执行的开始到结束都从源码分析完了。

AsyncTask的串行和并行

从上述源码分析中分析得到，默认情况下AsyncTask的执行效果是串行的，因为有了SerialExecutor类来维持保证队列的串行。如果想使用并行执行任务，那么可以直接跳过SerialExecutor类，使用executeOnExecutor()来执行任务。

四、AsyncTask使用不当的后果

1.)生命周期

AsyncTask不与任何组件绑定生命周期，所以在Activity/或者Fragment中创建执行AsyncTask时，最好在Activity/Fragment的onDestory()调用 cancel(boolean)；

2.)内存泄漏

如果AsyncTask被声明为Activity的非静态的内部类，那么AsyncTask会保留一个对创建了AsyncTask的Activity的引用。如果Activity已经被销毁，AsyncTask的后台线程还在执行，它将继续在内存里保留这个引用，导致Activity无法被回收，引起内存泄露。

3.) 结果丢失

屏幕旋转或Activity在后台被系统杀掉等情况会导致Activity的重新创建，之前运行的AsyncTask（非静态的内部类）会持有一个之前Activity的引用，这个引用已经无效，这时调用onPostExecute()再去更新界面将不再生效。

我们知道在**Android**系统中，我们执行完耗时操作都要另外开启子线程来执行，执行完线程以后线程会自动销毁。想象一下如果我们在项目中经常要执行耗时操作，如果经常要开启线程，接着又销毁线程，这无疑是很消耗性能的？那有什么解决方法呢？

1. 使用线程池
2. 使用HandlerThread

本篇文章主要讲解一下问题

1. HandlerThread的使用场景以及怎样使用HandlerThread？
2. HandlerThread源码分析

HandlerThread的使用场景以及怎样使用HandlerThread？

使用场景

HandlerThread是Google帮我们封装好的，可以用来执行多个耗时操作，而不需要多次开启线程，里面是采用Handler和Looper实现的。

> Handy class for starting a new thread that has a looper. The looper can then be used to create handler classes. Note that start() must still be called.

怎样使用HandlerThread？

1. 创建HandlerThread的实例对象

```
HandlerThread handlerThread = new HandlerThread("myHandlerThread"
);
```

该参数表示线程的名字，可以随便选择。

1. 启动我们创建的HandlerThread线程

```
handlerThread.start();
```

1. 将我们的handlerThread与Handler绑定在一起。 还记得是怎样将Handler与线程对象绑定在一起的吗？其实很简单，就是将线程的looper与Handler绑定在一起，代码如下：

```
mThreadHandler = new Handler(mHandlerThread.getLooper()) {
    @Override
    public void handleMessage(Message msg) {
        checkForUpdate();
        if(isUpdate){
            mThreadHandler.sendEmptyMessage(MSG_UPDATE_INFO);
        }
    }
};
```

注意必须按照以上三个步骤来，下面在讲解源码的时候会分析其原因

完整测试代码如下

```
public class MainActivity extends AppCompatActivity {
    private static final int MSG_UPDATE_INFO = 0x100;
    Handler mMainHandler = new Handler();
    private TextView mTv;
    private Handler mThreadHandler;
    private HandlerThread mHandlerThread;
    private boolean isUpdate = true;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        mTv = (TextView) findViewById(R.id.tv);
        initHandlerThread();
    }

    private void initHandlerThread() {
```

```
        mHandlerThread = new HandlerThread("xujun");
        mHandlerThread.start();
        mThreadHandler = new Handler(mHandlerThread.getLooper())
 {
            @Override
            public void handleMessage(Message msg) {
                checkForUpdate();
                if (isUpdate) {
                    mThreadHandler.sendEmptyMessage(MSG_UPDATE_I
NFO);
                }
            }
        };
    }

    /**
     * 模拟从服务器解析数据
     */
    private void checkForUpdate() {
        try {
            //模拟耗时
            Thread.sleep(1200);
            mMainHandler.post(new Runnable() {
                @Override
                public void run() {
                    String result = "实时更新中，当前股票行情：<font
 color='red'>%d</font>";
                    result = String.format(result, (int) (Math.r
andom() * 5000 + 1000));
                    mTv.setText(Html.fromHtml(result));
                }
            });
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    @Override
    protected void onResume() {
        isUpdate = true;
```

```
        super.onResume();
        mThreadHandler.sendEmptyMessage(MSG_UPDATE_INFO);
    }

    @Override
    protected void onPause() {
        super.onPause();
        isUpdate = false;
        mThreadHandler.removeMessages(MSG_UPDATE_INFO);
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        mHandlerThread.quit();
        mMainHandler.removeCallbacksAndMessages(null);
    }
}
```

运行以上测试代码，将可以看到如下效果图(例子不太恰当，主要使用场景是在**handleMessage**中执行耗时操作)

实时更新中，当前股票行情：1951

HandlerThread源码分析

官方源代码如下，是基于sdk23的，可以看到，只有一百多行代码而已。

```
public class HandlerThread extends Thread {
```

```
    int mPriority;
    int mTid = -1;
    Looper mLooper;

    public HandlerThread(String name) {
        super(name);
        mPriority = Process.THREAD_PRIORITY_DEFAULT;
    }

    public HandlerThread(String name, int priority) {
        super(name);
        mPriority = priority;
    }

    /**
     * Call back method that can be explicitly overridden if nee
ded to execute some
     * setup before Looper loops.
     */
    protected void onLooperPrepared() {
    }

    @Override
    public void run() {
        mTid = Process.myTid();
        Looper.prepare();
        //持有锁机制来获得当前线程的Looper对象
        synchronized (this) {
            mLooper = Looper.myLooper();
            //发出通知，当前线程已经创建mLooper对象成功，这里主要是通知g
etLooper方法中的wait
            notifyAll();
        }
        //设置线程的优先级别
        Process.setThreadPriority(mPriority);
        //这里默认是空方法的实现，我们可以重写这个方法来做一些线程开始之前
的准备，方便扩展
        onLooperPrepared();
        Looper.loop();
        mTid = -1;
```

```
    }

    public Looper getLooper() {
        if (!isAlive()) {
            return null;
        }
        // 直到线程创建完Looper之后才能获得Looper对象，Looper未创建成功
，阻塞
        synchronized (this) {
            while (isAlive() && mLooper == null) {
                try {
                    wait();
                } catch (InterruptedException e) {
                }
            }
        }
        return mLooper;
    }

    public boolean quit() {
        Looper looper = getLooper();
        if (looper != null) {
            looper.quit();
            return true;
        }
        return false;
    }

    public boolean quitSafely() {
        Looper looper = getLooper();
        if (looper != null) {
            looper.quitSafely();
            return true;
        }
        return false;
    }

    /**
     * Returns the identifier of this thread. See Process.myTid(
).
```

```
     */
    public int getThreadId() {
        return mTid;
    }
}
```

1）首先我们先来看一下它的构造方法

```
public HandlerThread(String name) {
    super(name);
    mPriority = Process.THREAD_PRIORITY_DEFAULT;
}

public HandlerThread(String name, int priority) {
    super(name);
    mPriority = priority;
}
```

有两个构造方法，一个参数的和两个参数的，name代表当前线程的名称，priority为线程的优先级别

2）接着我们来看一下run()方法，在run方法里面我们可以看到我们会初始化一个Looper，并设置线程的优先级别

```
public void run() {
    mTid = Process.myTid();
    Looper.prepare();
    //持有锁机制来获得当前线程的Looper对象
    synchronized (this) {
        mLooper = Looper.myLooper();
        //发出通知，当前线程已经创建mLooper对象成功，这里主要是通知getLo
oper方法中的wait
        notifyAll();
    }
    //设置线程的优先级别
    Process.setThreadPriority(mPriority);
    //这里默认是空方法的实现，我们可以重写这个方法来做一些线程开始之前的准备
，方便扩展
    onLooperPrepared();
    Looper.loop();
    mTid = -1;
}
```

- 还记得我们前面我们说到使用HandlerThread的时候必须调用 `start()` 方法，接着才可以将我们的HandlerThread和我们的handler绑定在一起吗?其实原因就是我们是在 `run()` 方法才开始初始化我们的looper，而我们调用HandlerThread的 `start()` 方法的时候，线程会交给虚拟机调度，由虚拟机自动调用run方法：

```
mHandlerThread.start();
mThreadHandler = new Handler(mHandlerThread.getLooper()) {
    @Override
    public void handleMessage(Message msg) {
        checkForUpdate();
        if(isUpdate){
            mThreadHandler.sendEmptyMessage(MSG_UPDATE_INFO);
        }
    }
};
```

- 这里我们为什么要使用锁机制和 `notifyAll()` ;，原因我们可以从 `getLooper()` 方法中知道

```
public Looper getLooper() {
    if (!isAlive()) {
        return null;
    }
    // 直到线程创建完Looper之后才能获得Looper对象，Looper未创建成功，阻塞

    synchronized (this) {
        while (isAlive() && mLooper == null) {
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
    }
    return mLooper;
}
```

总结：在获得**mLooper**对象的时候存在一个同步的问题，只有当线程创建成功并且**Looper**对象也创建成功之后才能获得**mLooper**的值。这里等待方法**wait**和**run**方法中的**notifyAll**方法共同完成同步问题。

3)接着我们来看一下quit方法和quitSafe方法

```
//调用这个方法退出Looper消息循环，及退出线程
public boolean quit() {
    Looper looper = getLooper();
    if (looper != null) {
        looper.quit();
        return true;
    }
    return false;
}
//调用这个方法安全地退出线程
@TargetApi(Build.VERSION_CODES.JELLY_BEAN_MR2)
public boolean quitSafely() {
    Looper looper = getLooper();
    if (looper != null) {
        looper.quitSafely();
        return true;
    }
    return false;
}
```

跟踪这两个方法容易知道只两个方法最终都会调用MessageQueue的 `quit(boolean safe)` 方法

```
void quit(boolean safe) {
    if (!mQuitAllowed) {
        throw new IllegalStateException("Main thread not allowed
 to quit.");
    }
    synchronized (this) {
        if (mQuitting) {
            return;
        }
        mQuitting = true;
        //安全退出调用这个方法
        if (safe) {
            removeAllFutureMessagesLocked();
        } else {//不安全退出调用这个方法
            removeAllMessagesLocked();
        }
        // We can assume mPtr != 0 because mQuitting was previou
sly false.
        nativeWake(mPtr);
    }
}
```

不安全的会调用 `removeAllMessagesLocked();` 这个方法，我们来看这个方法是怎样处理的，其实就是遍历Message链表，移除所有信息的回调，并重置为null。

```
private void removeAllMessagesLocked() {
    Message p = mMessages;
    while (p != null) {
        Message n = p.next;
        p.recycleUnchecked();
        p = n;
    }
    mMessages = null;
}
```

安全地会调用 `removeAllFutureMessagesLocked();` 这个方法，它会根据Message.when这个属性，判断我们当前消息队列是否正在处理消息，没有正在处理消息的话，直接移除所有回调，正在处理的话，等待该消息处理处理完毕再退出

该循环。因此说 `quitSafe()` 是安全的，而 `quit()` 方法是不安全的，因为quit方法不管是否正在处理消息，直接移除所有回调。

```
private void removeAllFutureMessagesLocked() {
    final long now = SystemClock.uptimeMillis();
    Message p = mMessages;
    if (p != null) {
        //判断当前队列中的消息是否正在处理这个消息，没有的话，直接移除所有
回调
        if (p.when > now) {
            removeAllMessagesLocked();
        } else {//正在处理的话，等待该消息处理处理完毕再退出该循环
            Message n;
            for (;;) {
                n = p.next;
                if (n == null) {
                    return;
                }
                if (n.when > now) {
                    break;
                }
                p = n;
            }
            p.next = null;
            do {
                p = n;
                n = p.next;
                p.recycleUnchecked();
            } while (n != null);
        }
    }
}
```

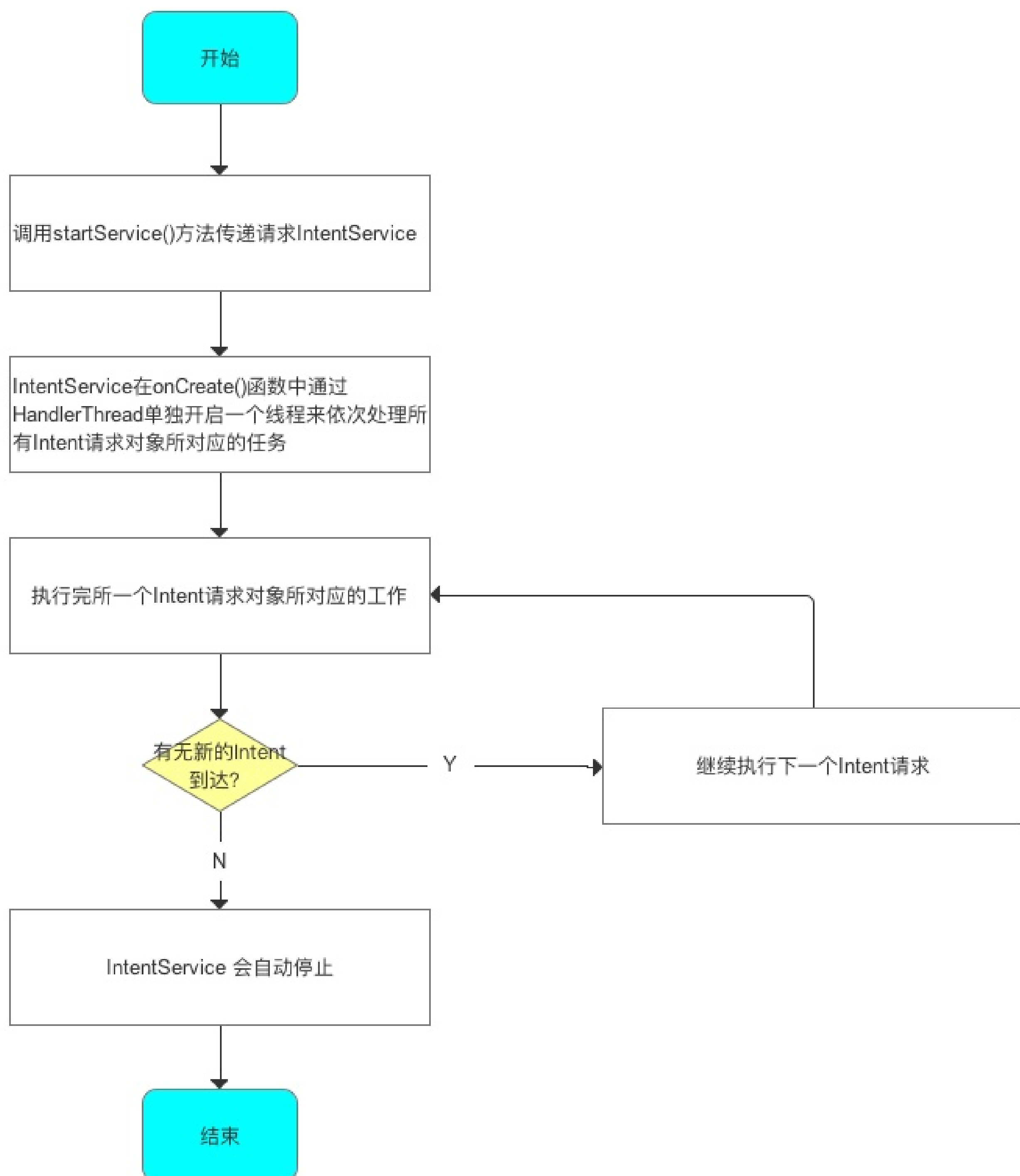
一、 定义

IntentService是Android里面的一个封装类，继承自四大组件之一的Service。

二、作用

处理异步请求，实现多线程。

三、 工作流程



注意：若启动IntentService多次，那么每个耗时操作则以队列的方式在IntentService的onHandleIntent回调方法中依次执行，执行完自动结束。

四、实现步骤

- 步骤1：定义IntentService的子类：传入线程名称、复写onHandleIntent()方法
- 步骤2：在Manifest.xml中注册服务

- 步骤3：在Activity中开启Service服务

五、具体实例

- 步骤1：定义IntentService的子类：传入线程名称、复写onHandleIntent()方法

```
package com.example.carson_ho.demoforintentservice;

import android.app.IntentService;
import android.content.Intent;
import android.util.Log;

/**
 * Created by Carson_Ho on 16/9/28.
 */
public class myIntentService extends IntentService {

    /*构造函数*/
    public myIntentService() {
        //调用父类的构造函数
        //构造函数参数=工作线程的名字
        super("myIntentService");

    }

    /*复写onHandleIntent()方法*/
    //实现耗时任务的操作
    @Override
    protected void onHandleIntent(Intent intent) {
        //根据Intent的不同进行不同的事务处理
        String taskName = intent.getExtras().getString("taskName"
);
        switch (taskName) {
            case "task1":
                Log.i("myIntentService", "do task1");
                break;
            case "task2":
                Log.i("myIntentService", "do task2");
                break;
```

```
            default:
                break;
        }
    }


    @Override
    public void onCreate() {
        Log.i("myIntentService", "onCreate");
        super.onCreate();
    }

    /*复写onStartCommand()方法*/
    //默认实现将请求的Intent添加到工作队列里
    @Override
    public int onStartCommand(Intent intent, int flags, int star
tId) {
        Log.i("myIntentService", "onStartCommand");
        return super.onStartCommand(intent, flags, startId);
    }

    @Override
    public void onDestroy() {
        Log.i("myIntentService", "onDestroy");
        super.onDestroy();
    }
}
```

- 步骤2：在Manifest.xml中注册服务

```
<service android:name=".myIntentService">
    <intent-filter>
        <action android:name="cn.scu.finch"/>
    </intent-filter>
</service>
```

- 步骤3：在Activity中开启Service服务

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

        //同一服务只会开启一个工作线程
        //在onHandleIntent函数里依次处理intent请求。

        Intent i = new Intent("cn.scu.finch");
        Bundle bundle = new Bundle();
        bundle.putString("taskName", "task1");
        i.putExtras(bundle);
        startService(i);

        Intent i2 = new Intent("cn.scu.finch");
        Bundle bundle2 = new Bundle();
        bundle2.putString("taskName", "task2");
        i2.putExtras(bundle2);
        startService(i2);

        startService(i);  //多次启动
    }
}
```

- 结果

Tag	Text
myIntentService	onCreate
myIntentService	onStartCommand
myIntentService	onStartCommand
myIntentService	do task1
myIntentService	onStartCommand
myIntentService	do task2
myIntentService	do task1
myIntentService	onDestroy

六、源码分析

接下来，我们会通过源码分析解决以下问题：

- IntentService如何单独开启一个新的工作线程；
- IntentService如何通过onStartCommand()传递给服务intent被依次插入到工作队列中

问题1：IntentService如何单独开启一个新的工作线程

```
// IntentService源码中的 onCreate() 方法
@Override
public void onCreate() {
    super.onCreate();
    // HandlerThread继承自Thread，内部封装了 Looper
    //通过实例化HandlerThread新建线程并启动
    //所以使用IntentService时不需要额外新建线程
    HandlerThread thread = new HandlerThread("IntentService[" +
mName + "]");
    thread.start();

    //获得工作线程的 Looper，并维护自己的工作队列
    mServiceLooper = thread.getLooper();
    //将上述获得Looper与新建的mServiceHandler进行绑定
    //新建的Handler是属于工作线程的。
    mServiceHandler = new ServiceHandler(mServiceLooper);
}

private final class ServiceHandler extends Handler {

    public ServiceHandler(Looper looper) {
        super(looper);
    }

    // IntentService的handleMessage方法把接收的消息交给onHandleInten
t()处理
    // onHandleIntent()是一个抽象方法，使用时需要重写的方法
    @Override
    public void handleMessage(Message msg) {
        // onHandleIntent 方法在工作线程中执行，执行完调用 stopSelf()
 结束服务。
        onHandleIntent((Intent) msg.obj);
        //onHandleIntent 处理完成后 IntentService会调用 stopSelf()
自动停止。
```

```
            stopSelf(msg.arg1);
        }
    }

    // onHandleIntent()是一个抽象方法，使用时需要重写的方法
    @WorkerThread
    protected abstract void onHandleIntent(Intent intent);
```

问题2：**IntentService**如何通过**onStartCommand()**传递给服务**intent**被依次插入到工作队列中

```
public int onStartCommand(Intent intent, int flags, int startId)
{
    onStart(intent, startId);
    return mRedelivery ? START_REDELIVER_INTENT : START_NOT_STIC
KY;
}

public void onStart(Intent intent, int startId) {
    Message msg = mServiceHandler.obtainMessage();
    msg.arg1 = startId;
    //把 intent 参数包装到 message 的 obj 中，然后发送消息，即添加到消
息队列里
    //这里的Intent 就是启动服务时startService(Intent) 里的 Intent。
    msg.obj = intent;
    mServiceHandler.sendMessage(msg);
}

//清除消息队列中的消息
@Override
public void onDestroy() {
    mServiceLooper.quit();
}
```

- 总结

从上面源码可以看出，IntentService本质是采用Handler & HandlerThread方式：

1. 通过HandlerThread单独开启一个名为**IntentService**的线程

2. 创建一个名叫ServiceHandler的内部Handler
3. 把内部Handler与HandlerThread所对应的子线程进行绑定
4. 通过onStartCommand()传递给服务intent，依次插入到工作队列中，并逐个发送给onHandleIntent()
5. 通过onHandleIntent()来依次处理所有Intent请求对象所对应的任务

因此我们通过复写方法onHandleIntent()，再在里面根据Intent的不同进行不同的线程操作就可以了

注意事项：工作任务队列是顺序执行的。

> 如果一个任务正在IntentService中执行，此时你再发送一个新的任务请求，这个新的任务会一直等待直到前面一个任务执行完毕才开始执行。

原因：

1. 由于onCreate() 方法只会调用一次，所以只会创建一个工作线程；
2. 当多次调用 startService(Intent) 时（onStartCommand也会调用多次）其实并不会创建新的工作线程，只是把消息加入消息队列中等待执行，所以，多次启动 **IntentService** 会按顺序执行事件；
3. 如果服务停止，会清除消息队列中的消息，后续的事件得不到执行。

七、使用场景

- 线程任务需要按顺序、在后台执行的使用场景

  > 最常见的场景：离线下载

- 由于所有的任务都在同一个Thread looper里面来做，所以不符合多个数据同时请求的场景。

八、对比

8.1 IntentService与Service的区别

- 从属性 & 作用上来说 Service：依赖于应用程序的主线程（不是独立的进程 or 线程）

  > 不建议在Service中编写耗时的逻辑和操作，否则会引起ANR；

IntentService：创建一个工作线程来处理多线程任务

- Service需要主动调用stopSelft()来结束服务，而IntentService不需要（在所有intent被处理完后，系统会自动关闭服务）

8.2 IntentService与其他线程的区别

- IntentService内部采用了HandlerThread实现，作用类似于后台线程；
- 与后台线程相比，IntentService是一种后台服务，优势是：优先级高（不容易被系统杀死），从而保证任务的执行。

> 对于后台线程，若进程中没有活动的四大组件，则该线程的优先级非常低，容易被系统杀死，无法保证任务的执行

一、Android中的缓存策略

一般来说，缓存策略主要包含缓存的添加、获取和删除这三类操作。如何添加和获取缓存这个比较好理解，那么为什么还要删除缓存呢？这是因为不管是内存缓存还是硬盘缓存，它们的缓存大小都是有限的。当缓存满了之后，再想其添加缓存，这个时候就需要删除一些旧的缓存并添加新的缓存。

因此LRU(Least Recently Used)缓存算法便应运而生，LRU是最近最少使用的算法，它的核心思想是当缓存满时，会优先淘汰那些最近最少使用的缓存对象。采用LRU算法的缓存有两种：LrhCache和DisLruCache，分别用于实现内存缓存和硬盘缓存，其核心思想都是LRU缓存算法。

二、LruCache的使用

LruCache是Android 3.1所提供的一个缓存类，所以在Android中可以直接使用LruCache实现内存缓存。而DisLruCache目前在Android 还不是Android SDK的一部分，但Android官方文档推荐使用该算法来实现硬盘缓存。

1.LruCache的介绍

LruCache是个泛型类，主要算法原理是把最近使用的对象用强引用（即我们平常使用的对象引用方式）存储在 LinkedHashMap 中。当缓存满时，把最近最少使用的对象从内存中移除，并提供了get和put方法来完成缓存的获取和添加操作。

2.LruCache的使用

LruCache的使用非常简单，我们就已图片缓存为例。

```
int maxMemory = (int) (Runtime.getRuntime().totalMemory() / 1024
);
int cacheSize = maxMemory / 8;
mMemoryCache = new LruCache<String, Bitmap>(cacheSize) {
    @Override
    protected int sizeOf(String key, Bitmap value) {
        return value.getRowBytes() * value.getHeight() / 1024;
    }
};
```

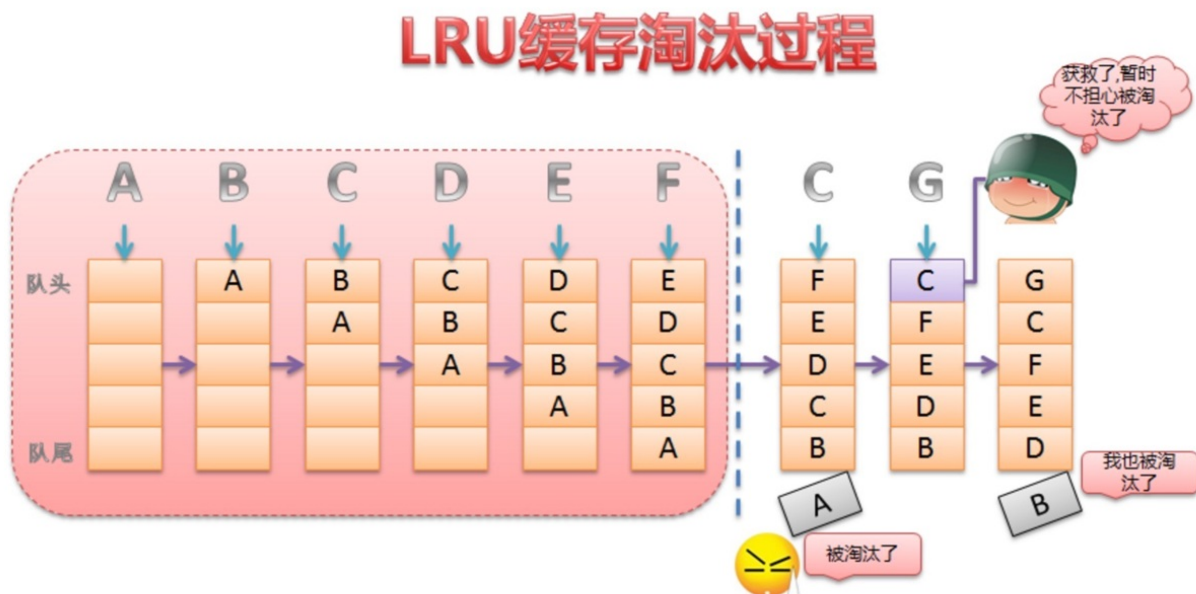
①设置LruCache缓存的大小，一般为当前进程可用容量的1/8。
②重写sizeOf方法，计算出要缓存的每张图片的大小。

注意：缓存的总容量和每个缓存对象的大小所用单位要一致。

三、LruCache的实现原理

LruCache的核心思想很好理解，就是要维护一个缓存对象列表，其中对象列表的排列方式是按照访问顺序实现的，即一直没访问的对象，将放在队尾，即将被淘汰。而最近访问的对象将放在队头，最后被淘汰。

如下图所示：



那么这个队列到底是由谁来维护的，前面已经介绍了是由LinkedHashMap来维护。

而LinkedHashMap是由数组+双向链表的数据结构来实现的。其中双向链表的结构可以实现访问顺序和插入顺序，使得LinkedHashMap中的对按照一定顺序排列起来。

通过下面构造函数来指定LinkedHashMap中双向链表的结构是访问顺序还是插入顺序。

```
public LinkedHashMap(int initialCapacity,
    float loadFactor,
    boolean accessOrder) {
    super(initialCapacity, loadFactor);
    this.accessOrder = accessOrder;
}
```

其中accessOrder设置为true则为访问顺序，为false，则为插入顺序。

以具体例子解释： 当设置为true时

```
public static final void main(String[] args) {
    LinkedHashMap<Integer, Integer> map = new LinkedHashMap<>(0,
0.75f, true);
    map.put(0, 0);
    map.put(1, 1);
    map.put(2, 2);
    map.put(3, 3);
    map.put(4, 4);
    map.put(5, 5);
    map.put(6, 6);
    map.get(1);
    map.get(2);

    for (Map.Entry<Integer, Integer> entry : map.entrySet()) {
        System.out.println(entry.getKey() + ":" + entry.getValue
());

    }
}
```

输出结果：

> 0:0
> 3:3
> 4:4
> 5:5
> 6:6
> 1:1
> 2:2

即最近访问的最后输出，那么这就正好满足的LRU缓存算法的思想。可见**LruCache**巧妙实现，就是利用了**LinkedHashMap**的这种数据结构。

下面我们在LruCache源码中具体看看，怎么应用LinkedHashMap来实现缓存的添加，获得和删除的。

```
public LruCache(int maxSize) {
    if (maxSize <= 0) {
        throw new IllegalArgumentException("maxSize <= 0");
    }
    this.maxSize = maxSize;
    this.map = new LinkedHashMap<K, V>(0, 0.75f, true);
}
```

从LruCache的构造函数中可以看到正是用了LinkedHashMap的访问顺序。

put()方法

```
public final V put(K key, V value) {
         //不可为空，否则抛出异常
    if (key == null || value == null) {
        throw new NullPointerException("key == null || value ==
null");
    }
    V previous;
    synchronized (this) {
            //插入的缓存对象值加1
        putCount++;
            //增加已有缓存的大小
        size += safeSizeOf(key, value);
           //向map中加入缓存对象
        previous = map.put(key, value);
            //如果已有缓存对象，则缓存大小恢复到之前
        if (previous != null) {
            size -= safeSizeOf(key, previous);
        }
    }
        //entryRemoved()是个空方法，可以自行实现
    if (previous != null) {
        entryRemoved(false, key, previous, value);
    }
        //调整缓存大小(关键方法)
    trimToSize(maxSize);
    return previous;
}
```

可以看到put()方法并没有什么难点，重要的就是在添加过缓存对象后，调用trimToSize()方法，来判断缓存是否已满，如果满了就要删除近期最少使用的算法。

trimToSize()方法

```
public void trimToSize(int maxSize) {
    //死循环
    while (true) {
        K key;
        V value;
        synchronized (this) {
            //如果map为空并且缓存size不等于0或者缓存size小于0，抛出异常
            if (size < 0 || (map.isEmpty() && size != 0)) {
                throw new IllegalStateException(getClass().getNa
me()
                        + ".sizeOf() is reporting inconsistent resul
ts!");
            }
            //如果缓存大小size小于最大缓存，或者map为空，不需要再删除缓存
对象，跳出循环
            if (size <= maxSize || map.isEmpty()) {
                break;
            }
            //迭代器获取第一个对象，即队尾的元素，近期最少访问的元素
            Map.Entry<K, V> toEvict = map.entrySet().iterator().
next();
            key = toEvict.getKey();
            value = toEvict.getValue();
            //删除该对象，并更新缓存大小
            map.remove(key);
            size -= safeSizeOf(key, value);
            evictionCount++;
        }
        entryRemoved(true, key, value, null);
    }
}
```

trimToSize()方法不断地删除LinkedHashMap中队尾的元素，即近期最少访问的，直到缓存大小小于最大值。

当调用LruCache的get()方法获取集合中的缓存对象时，就代表访问了一次该元素，将会更新队列，保持整个队列是按照访问顺序排序。这个更新过程就是在LinkedHashMap中的get()方法中完成的。

先看LruCache的get()方法

get()方法

```
public final V get(K key) {
        //key为空抛出异常
    if (key == null) {
        throw new NullPointerException("key == null");
    }

    V mapValue;
    synchronized (this) {
            //获取对应的缓存对象
            //get()方法会实现将访问的元素更新到队列头部的功能
        mapValue = map.get(key);
        if (mapValue != null) {
            hitCount++;
            return mapValue;
        }
        missCount++;
    }
```

其中LinkedHashMap的get()方法如下：

```
public V get(Object key) {
    LinkedHashMapEntry<K,V> e = (LinkedHashMapEntry<K,V>)getEntr
y(key);
    if (e == null)
        return null;
    //实现排序的关键方法
    e.recordAccess(this);
    return e.value;
}
```

调用recordAccess()方法如下：

```
void recordAccess(HashMap<K,V> m) {
    LinkedHashMap<K,V> lm = (LinkedHashMap<K,V>)m;
    //判断是否是访问排序
    if (lm.accessOrder) {
        lm.modCount++;
        //删除此元素
        remove();
        //将此元素移动到队列的头部
        addBefore(lm.header);
    }
}
```

由此可见**LruCache**中维护了一个集合**LinkedHashMap**，该**LinkedHashMap**是以访问顺序排序的。当调用**put()**方法时，就会在结合中添加元素，并调用**trimToSize()**判断缓存是否已满，如果满了就用**LinkedHashMap**的迭代器删除队尾元素，即近期最少访问的元素。当调用**get()**方法访问缓存对象时，就会调用**LinkedHashMap**的**get()**方法获得对应集合元素，同时会更新该元素到队头。

一、职能简介

Activity

Activity并不负责视图控制，它只是控制生命周期和处理事件。真正控制视图的是Window。一个Activity包含了一个Window，Window才是真正代表一个窗口。**Activity**就像一个控制器，统筹视图的添加与显示，以及通过其他回调方法，来与**Window**、以及**View**进行交互。

Window

Window是视图的承载器，内部持有一个 DecorView，而这个DecorView才是 view的根布局。Window是一个抽象类，实际在Activity中持有的是其子类PhoneWindow。PhoneWindow中有个内部类DecorView，通过创建DecorView来加载Activity中设置的布局 `R.layout.activity_main` 。Window 通过WindowManager将DecorView加载其中，并将DecorView交给ViewRoot，进行视图绘制以及其他交互。

DecorView

DecorView是FrameLayout的子类，它可以被认为是Android视图树的根节点视图。DecorView作为顶级View，一般情况下它内部包含一个竖直方向的LinearLayout，在这个**LinearLayout**里面有上下三个部分，上面是个**ViewStub**，延迟加载的视图（应该是设置**ActionBar**，根据**Theme**设置），中间的是标题栏**(**根据**Theme**设置，有的布局没有**)**，下面的是内容栏。 具体情况和Android版本及主体有关，以其中一个布局为例，如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
android"
    android:fitsSystemWindows="true"
    android:orientation="vertical">
    <!-- Popout bar for action modes -->
    <ViewStub
        android:id="@+id/action_mode_bar_stub"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:inflatedId="@+id/action_mode_bar"
        android:layout="@layout/action_mode_bar"
        android:theme="?attr/actionBarTheme" />

    <FrameLayout
        style="?android:attr/windowTitleBackgroundStyle"
        android:layout_width="match_parent"
        android:layout_height="?android:attr/windowTitleSize">

        <TextView
            android:id="@android:id/title"
            style="?android:attr/windowTitleStyle"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:background="@null"
            android:fadingEdge="horizontal"
            android:gravity="center_vertical" />
    </FrameLayout>

    <FrameLayout
        android:id="@android:id/content"
        android:layout_width="match_parent"
        android:layout_height="0dip"
        android:layout_weight="1"
        android:foreground="?android:attr/windowContentOverlay"
        android:foregroundGravity="fill_horizontal|top" />
</LinearLayout>
```

在Activity中通过setContentView所设置的布局文件其实就是被加到内容栏之中的，成为其唯一子View，就是上面的id为content的FrameLayout中，在代码中可以通过content来得到对应加载的布局。

```
ViewGroup content = (ViewGroup)findViewById(android.R.id.content
);
ViewGroup rootView = (ViewGroup) content.getChildAt(0);
```

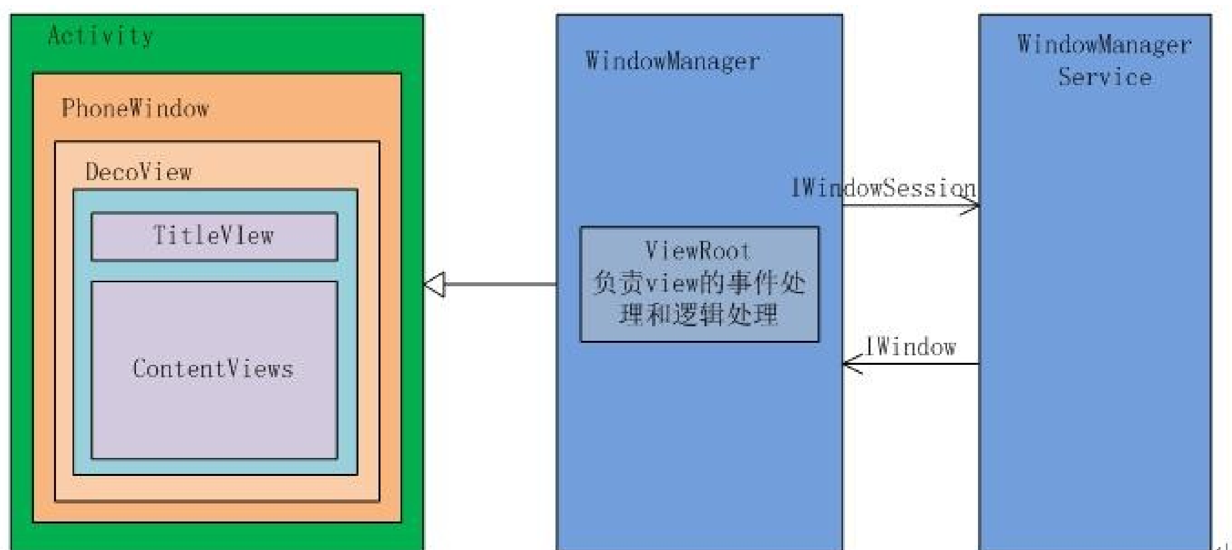
ViewRoot

ViewRoot可能比较陌生，但是其作用非常重大。所有View的绘制以及事件分发等交互都是通过它来执行或传递的。

ViewRoot对应**ViewRootImpl**类，它是连接**WindowManagerService**和**DecorView**的纽带，View的三大流程（测量（measure），布局（layout），绘制（draw））均通过ViewRoot来完成。

ViewRoot并不属于View树的一份子。从源码实现上来看，它既非View的子类，也非View的父类，但是，它实现了ViewParent接口，这让它可以作为View的名义上的父视图。RootView继承了Handler类，可以接收事件并分发，Android的所有触屏事件、按键事件、界面刷新等事件都是通过ViewRoot进行分发的。

下面结构图可以清晰的揭示四者之间的关系：



二、DecorView的创建

这部分内容主要讲DecorView是怎么一层层嵌套在Actvity，PhoneWindow中的，以及DecorView如何加载内部布局。

setContentView

先是从Activity中的setContentView()开始。

```
public void setContentView(@LayoutRes int layoutResID) {
    getWindow().setContentView(layoutResID);
    initWindowDecorActionBar();
}
```

可以看到实际是交给Window装载视图。下面来看看**Activity**是怎么获得**Window**对象的？

```
final void attach(Context context, ActivityThread aThread,
    Instrumentation instr, IBinder token, int ident,
    Application application, Intent intent, ActivityInfo info,
    CharSequence title, Activity parent, String id,
    NonConfigurationInstances lastNonConfigurationInstances,
    Configuration config, String referrer, IVoiceInteractor voic
eInteractor,
    Window window) {
        ........................................................
..........
        mWindow = new PhoneWindow(this, window);//创建一个Window对象

        mWindow.setWindowControllerCallback(this);
        mWindow.setCallback(this);//设置回调，向Activity分发点击或状
态改变等事件
        mWindow.setOnWindowDismissedCallback(this);
        ........................................................
.........
        mWindow.setWindowManager(
            (WindowManager)context.getSystemService(Context.WIND
OW_SERVICE),
            mToken, mComponent.flattenToString(),
                (info.flags & ActivityInfo.FLAG_HARDWARE_ACCELER
ATED) != 0);//给Window设置WindowManager对象
        ........................................................
............
}
```

在Activity中的attach()方法中，生成了PhoneWindow实例。既然有了Window对象，那么我们就可以设置**DecorView**给**Window**对象了。

```
public void setContentView(int layoutResID) {
    if (mContentParent == null) {//mContentParent为空，创建一个DecroView
        installDecor();
    } else {
        mContentParent.removeAllViews();//mContentParent不为空，删除其中的View
    }
    mLayoutInflater.inflate(layoutResID, mContentParent);//为mContentParent添加子View,即Activity中设置的布局文件
    final Callback cb = getCallback();
    if (cb != null && !isDestroyed()) {
        cb.onContentChanged();//回调通知，内容改变
    }
}
```

看了下来，可能有一个疑惑：**mContentParent**到底是什么？ 就是前面布局中 `@android:id/content` 所对应的FrameLayout。

通过上面的流程我们大致可以了解先在**PhoneWindow**中创建了一个**DecroView**，其中创建的过程中可能根据**Theme**不同，加载不同的布局格式，例如有没有**Title**，或有没有**ActionBar**等，然后再向**mContentParent**中加入子**View**，即**Activity**中设置的布局。到此位置，视图一层层嵌套添加上了。

下面具体来看看 `installDecor();` 方法，怎么创建的**DecroView**，并设置其整体布局？

```
private void installDecor() {
    if (mDecor == null) {
        mDecor = generateDecor(); //生成DecorView
        mDecor.setDescendantFocusability(ViewGroup.FOCUS_AFTER_D
ESCENDANTS);
        mDecor.setIsRootNamespace(true);
        if (!mInvalidatePanelMenuPosted && mInvalidatePanelMenuF
eatures != 0) {
            mDecor.postOnAnimation(mInvalidatePanelMenuRunnable)
;
        }
    }
    if (mContentParent == null) {
        mContentParent = generateLayout(mDecor); // 为DecorView设
置布局格式，并返回mContentParent
        ...
        }
    }
}
```

再来看看 generateDecor()

```
protected DecorView generateDecor() {
    return new DecorView(getContext(), -1);
}
```

很简单，创建了一个DecorView。

再看generateLayout

```
protected ViewGroup generateLayout(DecorView decor) {
    // 从主题文件中获取样式信息
    TypedArray a = getWindowStyle();

    ...................

    if (a.getBoolean(R.styleable.Window_windowNoTitle, false)) {
        requestFeature(FEATURE_NO_TITLE);
```

```
    } else if (a.getBoolean(R.styleable.Window_windowActionBar,
false)) {
        // Don't allow an action bar if there is no title.
        requestFeature(FEATURE_ACTION_BAR);
    }

    ................

    // 根据主题样式，加载窗口布局
    int layoutResource;
    int features = getLocalFeatures();
    // System.out.println("Features: 0x" + Integer.toHexString(f
eatures));
    if ((features & (1 << FEATURE_SWIPE_TO_DISMISS)) != 0) {
        layoutResource = R.layout.screen_swipe_dismiss;
    } else if(...){
        ...
    }

    View in = mLayoutInflater.inflate(layoutResource, null);//加
载layoutResource

    //往DecorView中添加子View，即文章开头介绍DecorView时提到的布局格式
，那只是一个例子，根据主题样式不同，加载不同的布局。
    decor.addView(in, new ViewGroup.LayoutParams(MATCH_PARENT, M
ATCH_PARENT));
    mContentRoot = (ViewGroup) in;

    ViewGroup contentParent = (ViewGroup)findViewById(ID_ANDROID
_CONTENT);// 这里获取的就是mContentParent
    if (contentParent == null) {
        throw new RuntimeException("Window couldn't find content
 container view");
    }

    if ((features & (1 << FEATURE_INDETERMINATE_PROGRESS)) != 0)
 {
        ProgressBar progress = getCircularProgressBar(false);
        if (progress != null) {
            progress.setIndeterminate(true);
```

```
        }
    }

    if ((features & (1 << FEATURE_SWIPE_TO_DISMISS)) != 0) {
        registerSwipeCallbacks();
    }

    // Remaining setup -- of background and title -- that only a
pplies
    // to top-level windows.
    ...

    return contentParent;
}
```

虽然比较复杂，但是逻辑还是很清楚的。先从主题中获取样式，然后根据样式，加载对应的布局到DecorView中，然后从中获取mContentParent。获得到之后，可以回到上面的代码，为mContentParent添加View，即Activity中的布局。

以上就是DecorView的创建过程，其实到 `installDecor()` 就已经介绍完了，后面只是具体介绍其中的逻辑。

三、DecorView的显示

以上仅仅是将DecorView建立起来。通过setContentView()设置的界面，为什么在onResume()之后才对用户可见呢？

这就要从ActivityThread开始说起。

```
private void handleLaunchActivity(ActivityClientRecord r, Intent
 customIntent) {

    //就是在这里调用了Activity.attach()呀，接着调用了Activity.onCreat
e()和Activity.onStart()生命周期，
    //但是由于只是初始化了mDecor，添加了布局文件，还没有把
    //mDecor添加到负责UI显示的PhoneWindow中，所以这时候对用户来说，是不
可见的
    Activity a = performLaunchActivity(r, customIntent);

    ......

    if (a != null) {
    //这里面执行了Activity.onResume()
    handleResumeActivity(r.token, false, r.isForward,
                        !r.activity.mFinished && !r.startsNotRes
umed);

    if (!r.activity.mFinished && r.startsNotResumed) {
        try {
                r.activity.mCalled = false;
                //执行Activity.onPause()
                mInstrumentation.callActivityOnPause(r.activity)
;
                }
        }
    }
}
```

重点看下handleResumeActivity(),在这其中，DecorView将会显示出来，同时重要的一个角色：ViewRoot也将登场。

```
final void handleResumeActivity(IBinder token, boolean clearHide
,
                                boolean isForward, boolean reall
yResume) {

    //这个时候，Activity.onResume()已经调用了，但是现在界面还是不可见的
    ActivityClientRecord r = performResumeActivity(token, clearH
```

```
ide);

    if (r != null) {
        final Activity a = r.activity;
        if (r.window == null && !a.mFinished && willBeVisible) {
            r.window = r.activity.getWindow();
            View decor = r.window.getDecorView();
            //decor对用户不可见
            decor.setVisibility(View.INVISIBLE);
            ViewManager wm = a.getWindowManager();
            WindowManager.LayoutParams l = r.window.getAttribute
s();
            a.mDecor = decor;

            l.type = WindowManager.LayoutParams.TYPE_BASE_APPLIC
ATION;

            if (a.mVisibleFromClient) {
                a.mWindowAdded = true;
                //被添加进WindowManager了，但是这个时候，还是不可见的
                wm.addView(decor, l);
            }

            if (!r.activity.mFinished && willBeVisible
                    && r.activity.mDecor != null && !r.hideForNo
w) {
                //在这里，执行了重要的操作,使得DecorView可见
                if (r.activity.mVisibleFromClient) {
                    r.activity.makeVisible();
                }
            }
        }
    }
}
```

当我们执行了Activity.makeVisible()方法之后，界面才对我们是可见的。

```
void makeVisible() {
   if (!mWindowAdded) {
        ViewManager wm = getWindowManager();
        wm.addView(mDecor, getWindow().getAttributes());//将Deco
rView添加到WindowManager
        mWindowAdded = true;
    }
    mDecor.setVisibility(View.VISIBLE);//DecorView可见
}
```

到此DecorView便可见，显示在屏幕中。但是在这其中, `wm.addView(mDecor, getWindow().getAttributes());` 起到了重要的作用，因为其内部创建了一个ViewRootImpl对象，负责绘制显示各个子View。

具体来看 `addView()` 方法，因为WindowManager是个接口，具体是交给WindowManagerImpl来实现的。

```
public final class WindowManagerImpl implements WindowManager {

    private final WindowManagerGlobal mGlobal = WindowManagerGlo
bal.getInstance();
    ...
    @Override
    public void addView(View view, ViewGroup.LayoutParams params)
 {
        mGlobal.addView(view, params, mDisplay, mParentWindow);
    }
}
```

交给WindowManagerGlobal 的addView()方法去实现

```
public void addView(View view, ViewGroup.LayoutParams params,
                    Display display, Window parentWindow) {

    final WindowManager.LayoutParams wparams = (WindowManager.La
youtParams) params;

    ......

    synchronized (mLock) {

        ViewRootImpl root;
        //实例化一个ViewRootImpl对象
        root = new ViewRootImpl(view.getContext(), display);
        view.setLayoutParams(wparams);

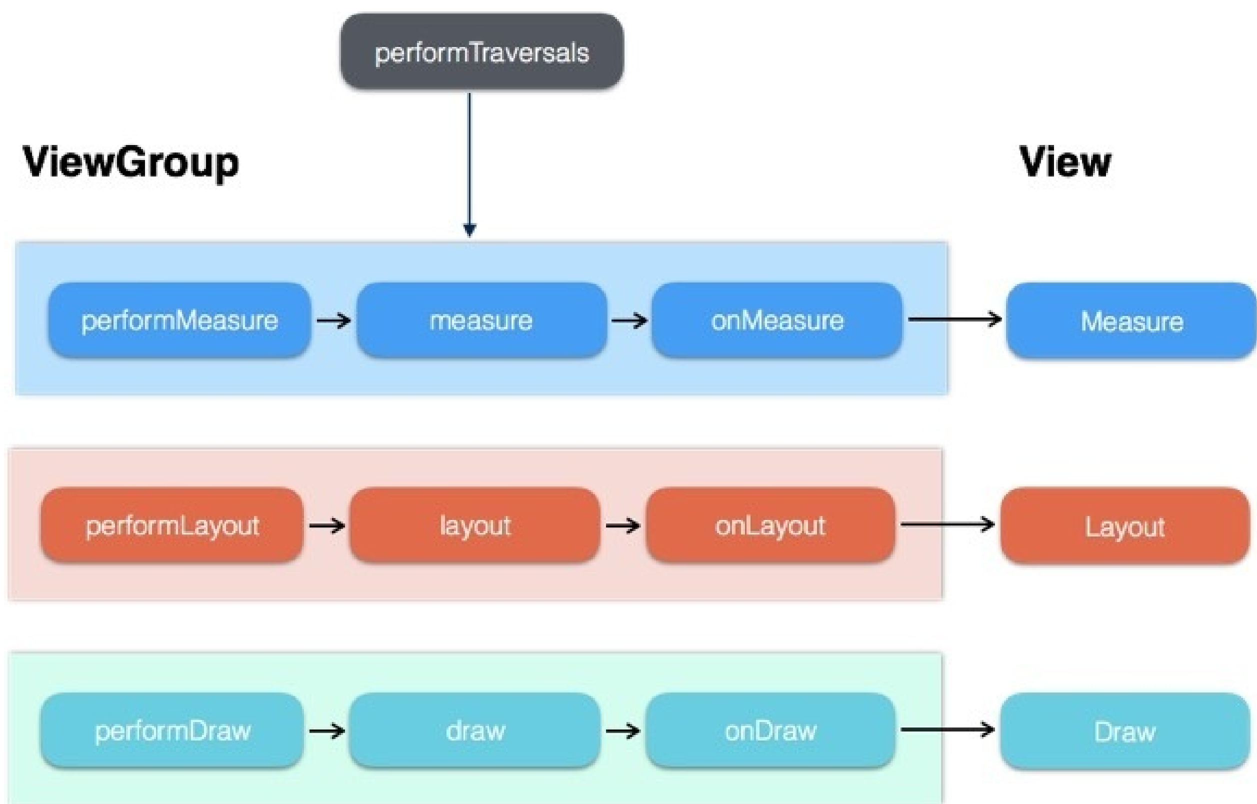
        mViews.add(view);
        mRoots.add(root);
        mParams.add(wparams);
    }

    ......

    try {
        //将DecorView交给ViewRootImpl
        root.setView(view, wparams, panelParentView);
    } catch (RuntimeException e) {

    }
}
```

看到其中实例化了ViewRootImpl对象，然后调用其setView()方法。其中setView()方法经过一些列折腾，最终调用了performTraversals()方法，然后依照下图流程层层调用，完成绘制，最终界面才显示出来。



其实ViewRootImpl的作用不止如此，还有许多功能，如事件分发。

要知道，当用户点击屏幕产生一个触摸行为，这个触摸行为则是通过底层硬件来传递捕获，然后交给ViewRootImpl，接着将事件传递给DecorView，而DecorView再交给PhoneWindow，PhoneWindow再交给Activity，然后接下来就是我们常见的View事件分发了。

硬件 **-> ViewRootImpl -> DecorView -> PhoneWindow -> Activity**

不详细介绍了，如果感兴趣，可以看这篇文章。

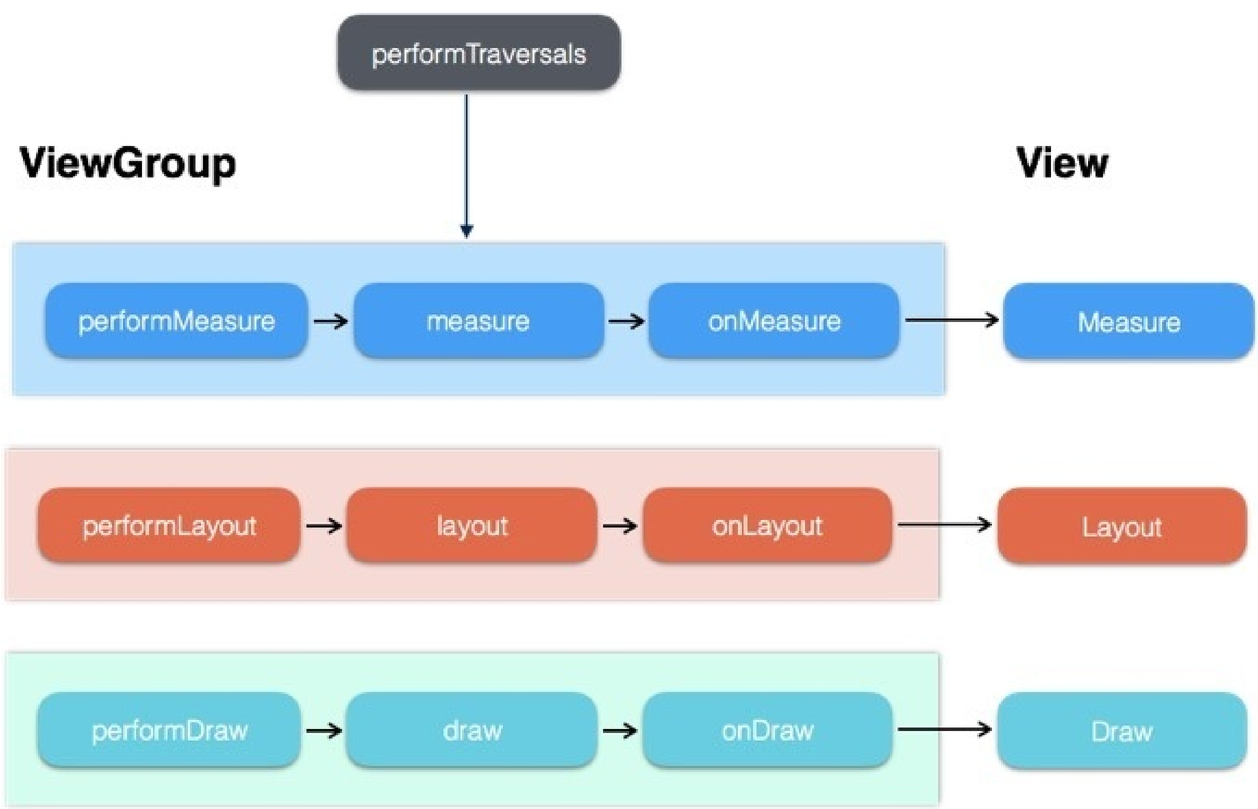
由此可见ViewRootImpl的重要性，是个连接器，负责WindowManagerService与DecorView之间的通信。

四、总结

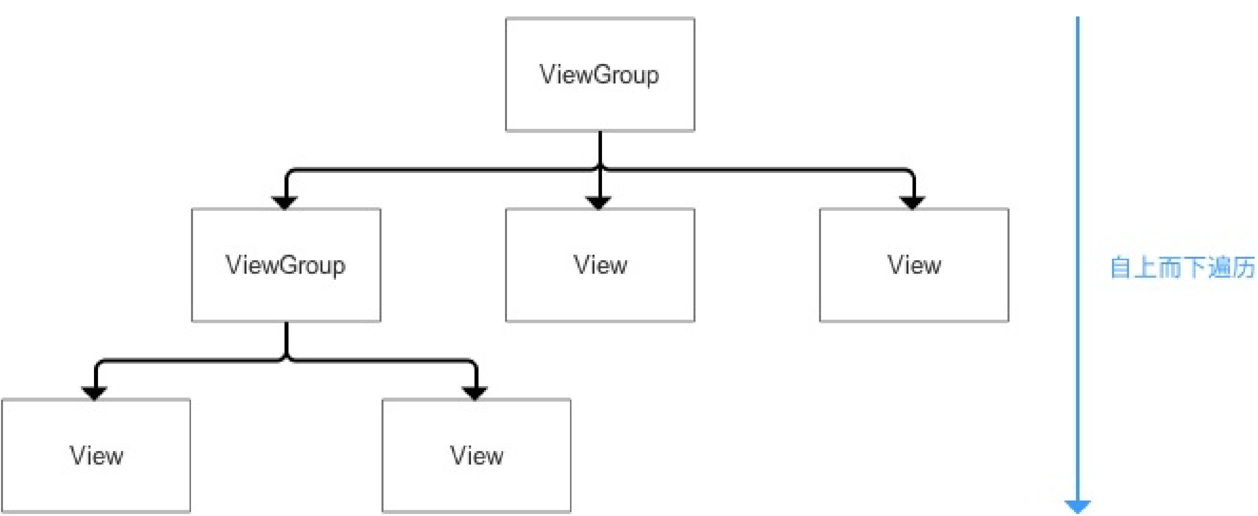
以上通过源码形式介绍了Window、Activity、DecorView以及ViewRoot之间的错综关系，以及如何创建并显示DecorView。

通过以上了解可以知道，**Activity**就像个控制器，不负责视图部分。**Window**像个承载器，装着内部视图。**DecorView**就是个顶层视图，是所有**View**的最外层布局。**ViewRoot**像个连接器，负责沟通，通过硬件的感知来通知视图，进行用户之间的交互。

一、View绘制的流程框架



View的绘制是从上往下一层层迭代下来的。DecorView-->ViewGroup（--->ViewGroup）-->View ，按照这个流程从上往下，依次measure(测量),layout(布局),draw(绘制)。



二、Measure流程

顾名思义，就是测量每个控件的大小。

调用measure()方法，进行一些逻辑处理，然后调用onMeasure()方法，在其中调用setMeasuredDimension()设定View的宽高信息，完成View的测量操作。

```
public final void measure(int widthMeasureSpec, int heightMeasureSpec) {
}
```

measure()方法中，传入了两个参数 widthMeasureSpec, heightMeasureSpec 表示View的宽高的一些信息。

```
protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
        setMeasuredDimension(getDefaultSize(getSuggestedMinimumWidth(), widthMeasureSpec),
                getDefaultSize(getSuggestedMinimumHeight(), heightMeasureSpec));
    }
```

由上述流程来看Measure流程很简单，关键点是在于widthMeasureSpec, heightMeasureSpec这两个参数信息怎么获得？

如果有了widthMeasureSpec, heightMeasureSpec，通过一定的处理(可以重写，自定义处理步骤)，从中获取View的宽/高，调用setMeasuredDimension()方法，指定View的宽高，完成测量工作。

MeasureSpec的确定

先介绍下什么是MeasureSpec？



MeasureSpec由两部分组成，一部分是测量模式，另一部分是测量的尺寸大小。

其中，Mode模式共分为三类

UNSPECIFIED ：不对View进行任何限制，要多大给多大，一般用于系统内部

EXACTLY：对应LayoutParams中的match_parent和具体数值这两种模式。检测到View所需要的精确大小，这时候View的最终大小就是SpecSize所指定的值，

AT_MOST ：对应LayoutParams中的wrap_content。View的大小不能大于父容器的大小。

那么**MeasureSpec**又是如何确定的？

对于DecorView，其确定是通过屏幕的大小，和自身的布局参数LayoutParams。

这部分很简单，根据LayoutParams的布局格式（match_parent，wrap_content或指定大小），将自身大小，和屏幕大小相比，设置一个不超过屏幕大小的宽高，以及对应模式。

对于其他View（包括ViewGroup），其确定是通过父布局的MeasureSpec和自身的布局参数LayoutParams。

这部分比较复杂。以下列图表表示不同的情况：

View的MeasureSpec创建规则

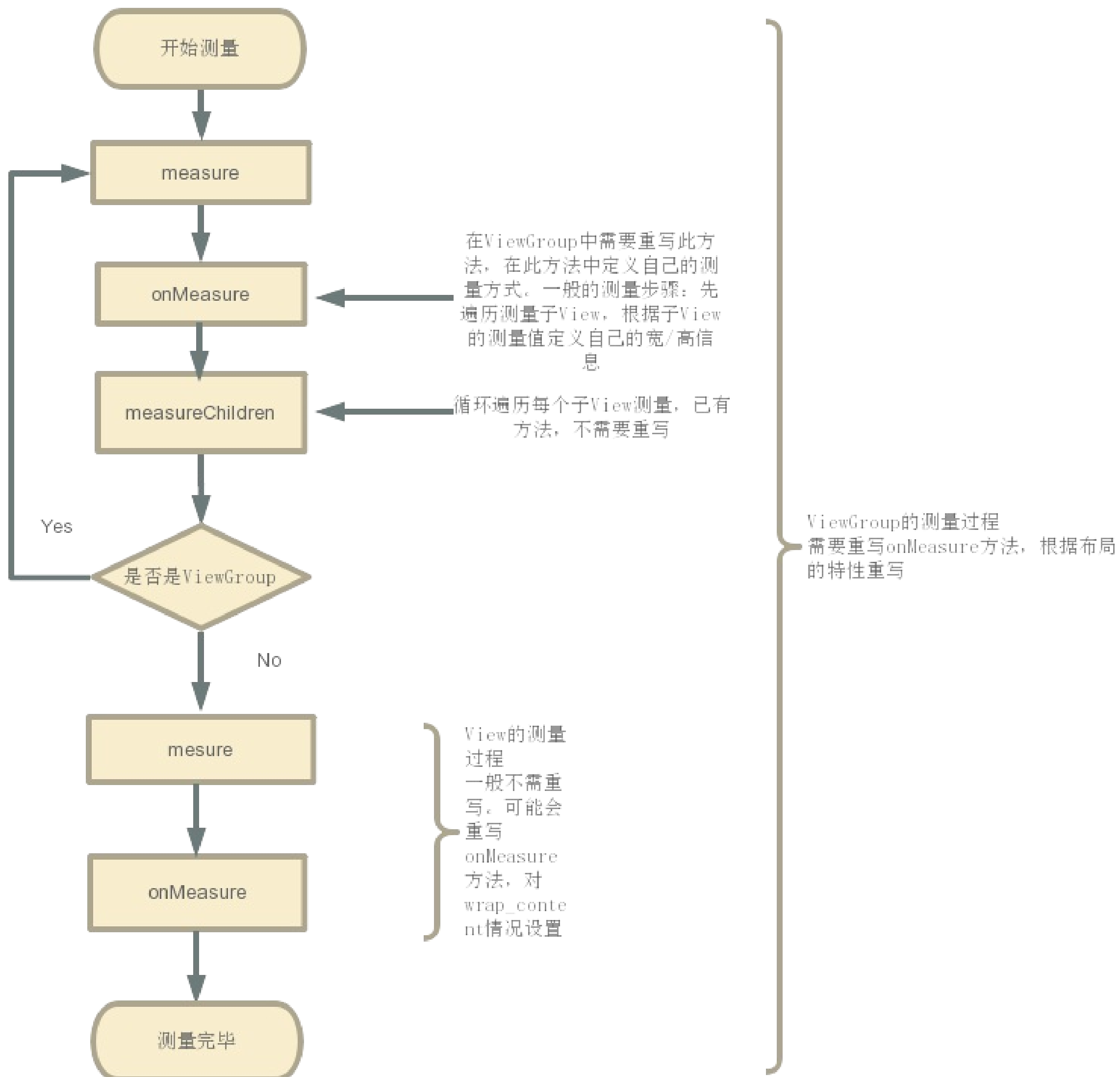
parentSpecMode / subView Size	EXACTLY	AT_MOST	UNSPECIFIED
具体大小(如100px)	childSize EXACTLY	childSize EXACTLY	childSize EXACTLY
MATCH_PARENT	parentLeftSize EXACTLY	parentLeftSize AT_MOST	0 UNSPECIFIED
WRAP_CONTENT	parentLeftSize AT_MOST	parentLeftSize AT_MOST	0 UNSPECIFIED

当子**View**的**LayoutParams**的布局格式是**wrap_content**，可以看到子**View**的大小是父**View**的剩余尺寸，和设置成**match_parent**时，子**View**的大小没有区别。为了显示区别，一般在自定义**View**时，需要重写**onMeasure**方法，处理**wrap_content**时的情况，进行特别指定。

从这里看出**MeasureSpec**的指定也是从顶层布局开始一层层往下去，父布局影响子布局。

可能关于MeasureSpec如何确定View大小还有些模糊，篇幅有限，没详细具体展开介绍，可以看这篇文章

View的测量流程：



三、Layout流程

测量完View大小后，就需要将View布局在Window中，View的布局主要通过确定上下左右四个点来确定的。

其中布局也是自上而下，不同的是**ViewGroup**先在**layout()**中确定自己的布局，然后在**onLayout()**方法中再调用子**View**的**layout()**方法，让子**View**布局。在**Measure**过程中，**ViewGroup**一般是先测量子**View**的大小，然后再确定自身的大小。

```
public void layout(int l, int t, int r, int b) {

    // 当前视图的四个顶点
    int oldL = mLeft;
    int oldT = mTop;
    int oldB = mBottom;
    int oldR = mRight;

    // setFrame（） / setOpticalFrame（）：确定View自身的位置
    // 即初始化四个顶点的值，然后判断当前View大小和位置是否发生了变化并返回

 boolean changed = isLayoutModeOptical(mParent) ?
           setOpticalFrame(l, t, r, b) : setFrame(l, t, r, b);

    //如果视图的大小和位置发生变化，会调用onLayout（）
    if (changed || (mPrivateFlags & PFLAG_LAYOUT_REQUIRED) == PF
LAG_LAYOUT_REQUIRED) {

        // onLayout（）：确定该View所有的子View在父容器的位置
        onLayout(changed, l, t, r, b);
  ...

}
```

上面看出通过 setFrame（） / setOpticalFrame（）：确定View自身的位置，通过onLayout()确定子View的布局。 setOpticalFrame（）内部也是调用了setFrame（），所以具体看setFrame（）怎么确定自身的位置布局。

```
protected boolean setFrame(int left, int top, int right, int bot
tom) {
    ...
// 通过以下赋值语句记录下了视图的位置信息，即确定View的四个顶点
// 即确定了视图的位置
    mLeft = left;
    mTop = top;
    mRight = right;
    mBottom = bottom;

    mRenderNode.setLeftTopRightBottom(mLeft, mTop, mRight, mBott
om);
}
```

确定了自身的位置后，就要通过onLayout()确定子View的布局。onLayout()是一个可继承的空方法。

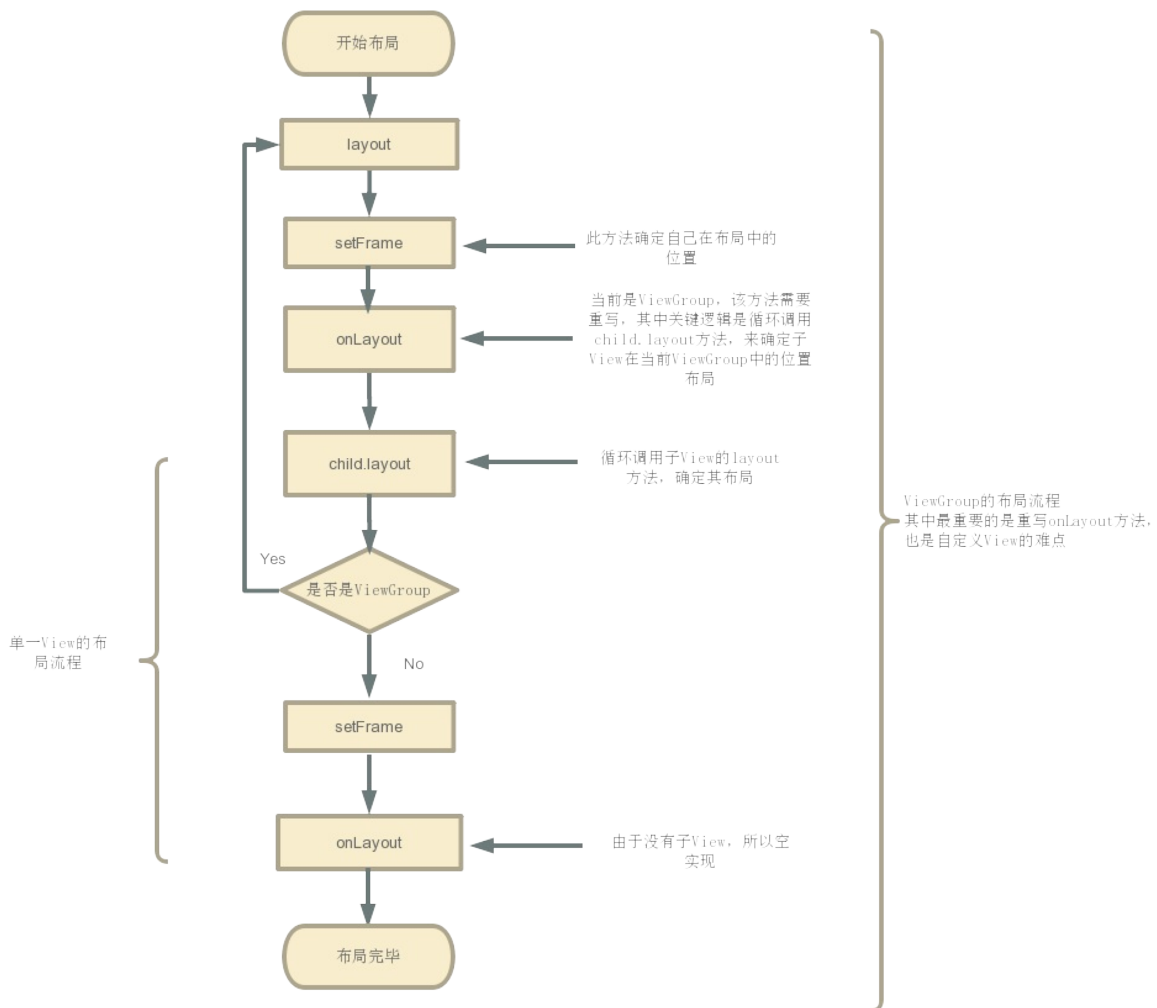
```
protected void onLayout(boolean changed, int left, int top, int
right, int bottom) {
    }
```

如果当前**View**就是一个单一的**View**，那么没有子**View**，就不需要实现该方法。

如果当前**View**是一个**ViewGroup**，就需要实现**onLayout**方法，该方法的实现个自定义**ViewGroup**时其特性有关，必须自己实现。

由此便完成了一层层的的布局工作。

View的布局流程：



四、Draw过程

View的绘制过程遵循如下几步：

①绘制背景 background.draw(canvas)

②绘制自己（onDraw）

③绘制Children(dispatchDraw)

④绘制装饰（onDrawScrollBars）

从源码中可以清楚地看出绘制的顺序。

```
public void draw(Canvas canvas) {
// 所有的视图最终都是调用 View 的 draw （）绘制视图（ ViewGroup 没有复写
此方法）
```

```
// 在自定义View时，不应该复写该方法，而是复写 onDraw(Canvas) 方法进行绘
制。
// 如果自定义的视图确实要复写该方法，那么需要先调用 super.draw(canvas)完
成系统的绘制，然后再进行自定义的绘制。
    ...
    int saveCount;
    if (!dirtyOpaque) {
          // 步骤1： 绘制本身View背景
        drawBackground(canvas);
    }

        // 如果有必要，就保存图层（还有一个复原图层）
        // 优化技巧：
        // 当不需要绘制 Layer 时，“保存图层“和“复原图层“这两步会跳过
        // 因此在绘制的时候，节省 layer 可以提高绘制效率
        final int viewFlags = mViewFlags;
        if (!verticalEdges && !horizontalEdges) {

        if (!dirtyOpaque)
             // 步骤2：绘制本身View内容  默认为空实现，  自定义View时需
要进行复写
            onDraw(canvas);

        ......
        // 步骤3：绘制子View   默认为空实现 单一View中不需要实现，ViewG
roup中已经实现该方法
        dispatchDraw(canvas);

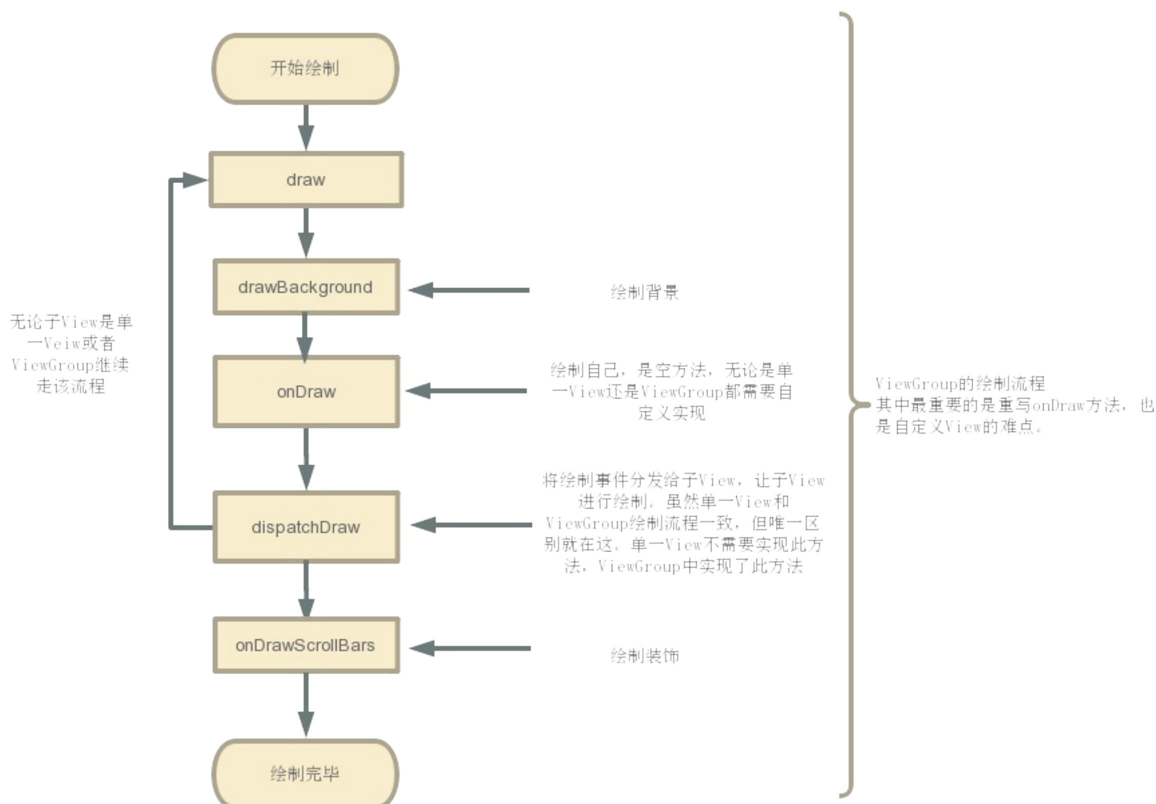
        ........

        // 步骤4：绘制滑动条和前景色等等
        onDrawScrollBars(canvas);

       ..........
        return;
    }
    ...
}
```

无论是**ViewGroup**还是单一的**View**，都需要实现这套流程，不同的是，在**ViewGroup**中，实现了 **dispatchDraw()**方法，而在单一子**View**中不需要实现该方法。自定义**View**一般要重写**onDraw()**方法，在其中绘制不同的样式。

View绘制流程：



五、总结

从View的测量、布局和绘制原理来看，要实现自定义View，根据自定义View的种类不同，可能分别要自定义实现不同的方法。但是这些方法不外乎：**onMeasure()**方法，**onLayout()**方法，**onDraw()**方法。

onMeasure()方法：单一View，一般重写此方法，针对wrap_content情况，规定View默认的大小值，避免于match_parent情况一致。ViewGroup，若不重写，就会执行和单子View中相同逻辑，不会测量子View。一般会重写onMeasure()方法，循环测量子View。

onLayout()方法:单一View，不需要实现该方法。ViewGroup必须实现，该方法是个抽象方法，实现该方法，来对子View进行布局。

onDraw()方法：无论单一View，或者ViewGroup都需要实现该方法，因其是个空方法

一、什么是Dalvik虚拟机

Dalvik是Google公司自己设计用于Android平台的Java虚拟机，它是Android平台的重要组成部分，支持dex格式（Dalvik Executable）的Java应用程序的运行。dex格式是专门为Dalvik设计的一种压缩格式，适合内存和处理器速度有限的系统。Google对其进行了特定的优化，使得Dalvik具有高效、简洁、节省资源的特点。从Android系统架构图知，Dalvik虚拟机运行在Android的运行时库层。

Dalvik作为面向Linux、为嵌入式操作系统设计的虚拟机，主要负责完成对象生命周期管理、堆栈管理、线程管理、安全和异常管理，以及垃圾回收等。另外，Dalvik早期并没有JIT编译器，直到Android2.2才加入了对JIT的技术支持。

二、Dalvik虚拟机的特点

体积小，占用内存空间小；

专有的DEX可执行文件格式，体积更小，执行速度更快；

常量池采用32位索引值，寻址类方法名，字段名，常量更快；

基于寄存器架构，并拥有一套完整的指令系统；

提供了对象生命周期管理，堆栈管理，线程管理，安全和异常管理以及垃圾回收等重要功能；

所有的Android程序都运行在Android系统进程里，每个进程对应着一个Dalvik虚拟机实例。

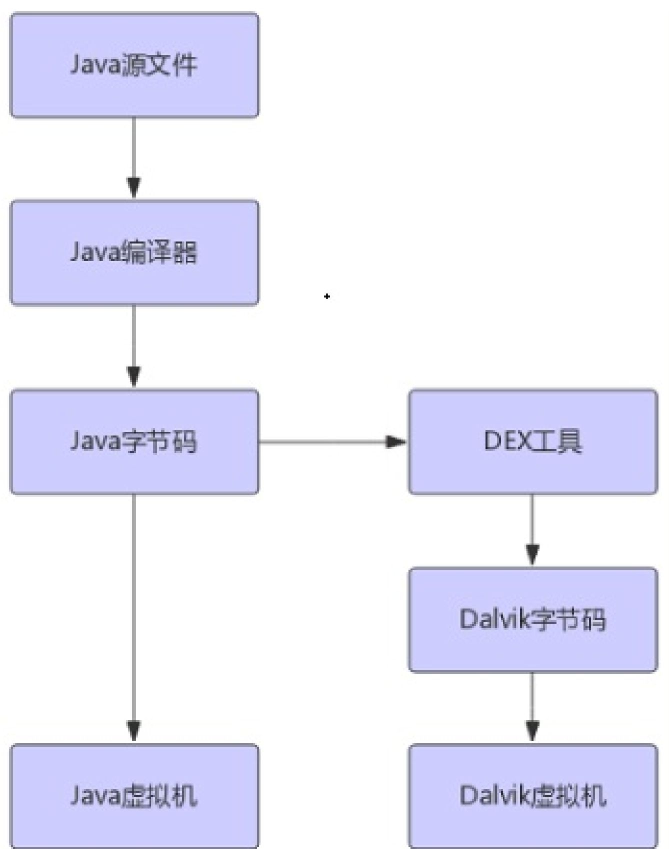
三、Dalvik虚拟机和Java虚拟机的区别

Dalvik虚拟机与传统的Java虚拟机有着许多不同点，两者并不兼容，它们显著的不同点主要表现在以下几个方面：

Java虚拟机运行的是**Java**字节码，**Dalvik**虚拟机运行的是**Dalvik**字节码。

传统的Java程序经过编译，生成Java字节码保存在class文件中，Java虚拟机通过解码class文件中的内容来运行程序。而Dalvik虚拟机运行的是Dalvik字节码，所有的Dalvik字节码由Java字节码转换而来，并被打包到一个DEX（Dalvik

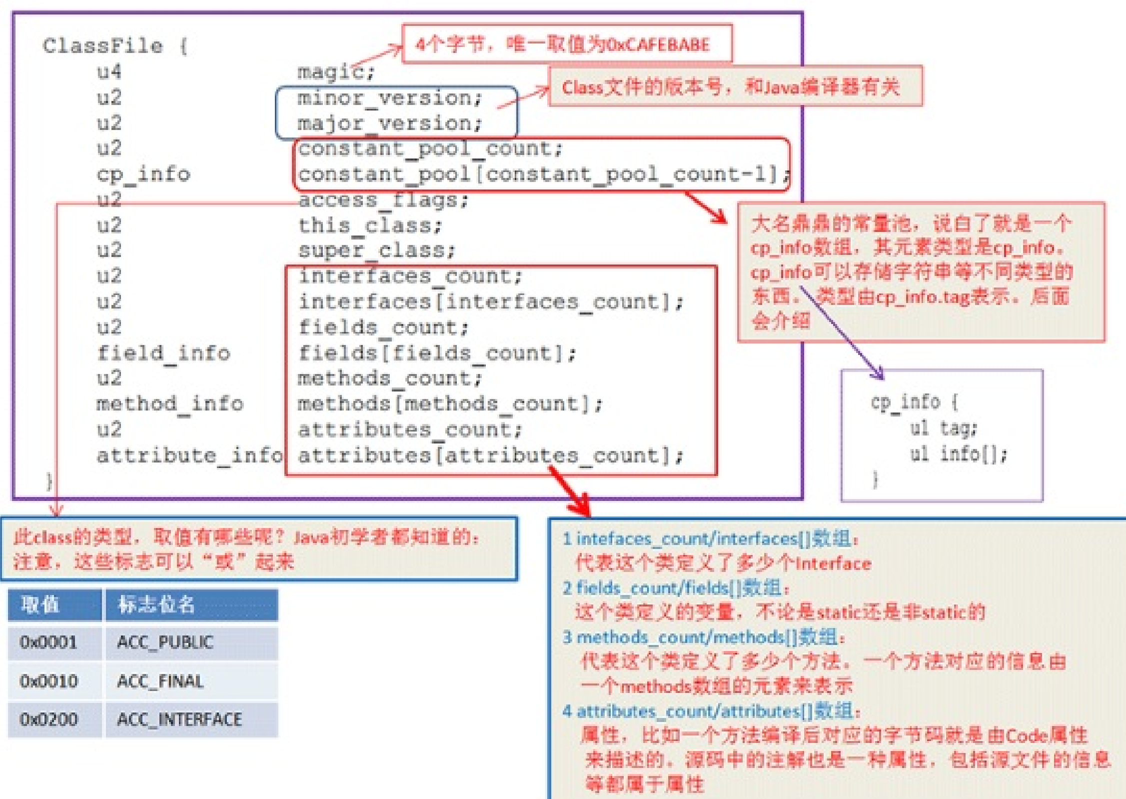
Executable）可执行文件中。Dalvik虚拟机通过解释DEX文件来执行这些字节码。



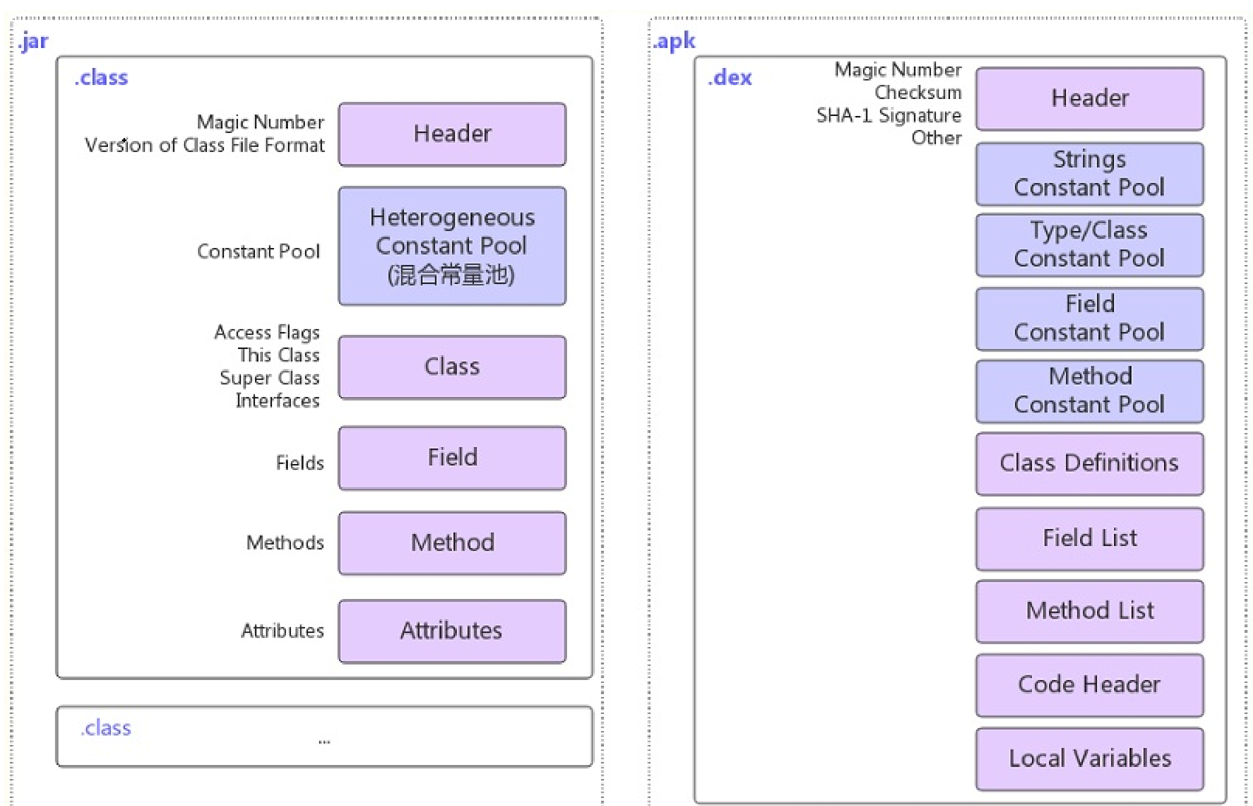
Dalvik可执行文件体积小。**Android SDK**中有一个叫**dx**的工具负责将**Java**字节码转换为**Dalvik**字节码。

dx工具对Java类文件重新排列，消除在类文件中出现的所有冗余信息，避免虚拟机在初始化时出现反复的文件加载与解析过程。一般情况下，Java类文件中包含多个不同的方法签名，如果其他的类文件引用该类文件中的方法，方法签名也会被复制到其类文件中，也就是说，多个不同的类会同时包含相同的方法签名，同样地，大量的字符串常量在多个类文件中也被重复使用。这些冗余信息会直接增加文件的体积，同时也会严重影响虚拟机解析文件的效率。消除其中的冗余信息，重新组合形成一个常量池，所有的类文件共享同一个常量池。由于**dx**工具对常量池的压缩，使得相同的字符串，常量在**DEX**文件中只出现一次，从而减小了文件的体积。

针对每个Class文件，都由如下格式进行组成：



dex格式文件使用共享的、特定类型的常量池机制来节省内存。常量池存储类中的所有字面常量，它包括字符串常量、字段常量等值。

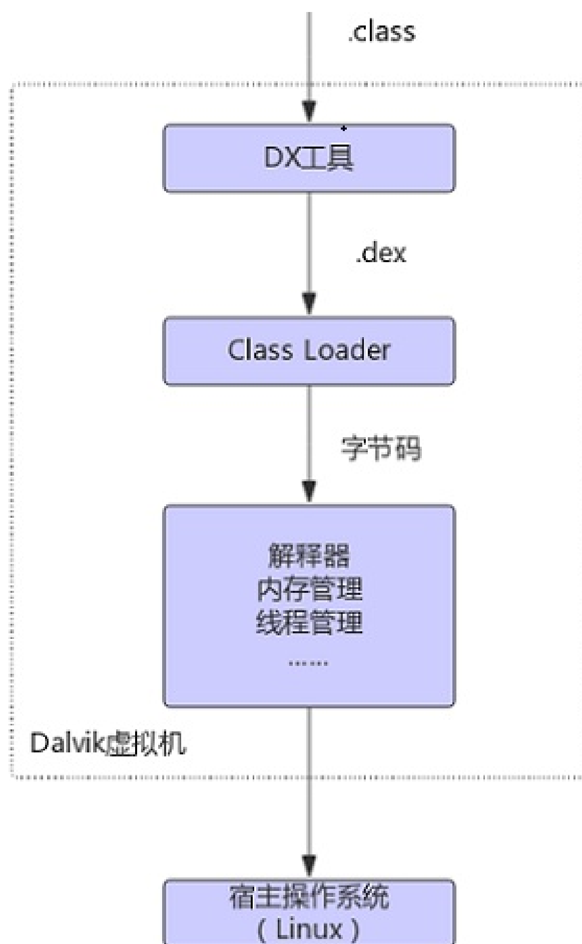


简单来讲，dex格式文件就是将多个class文件中公有的部分统一存放，去除冗余信息。

Java虚拟机与**Dalvik**虚拟机架构不同。这也是Dalvik与JVM之间最大的区别。

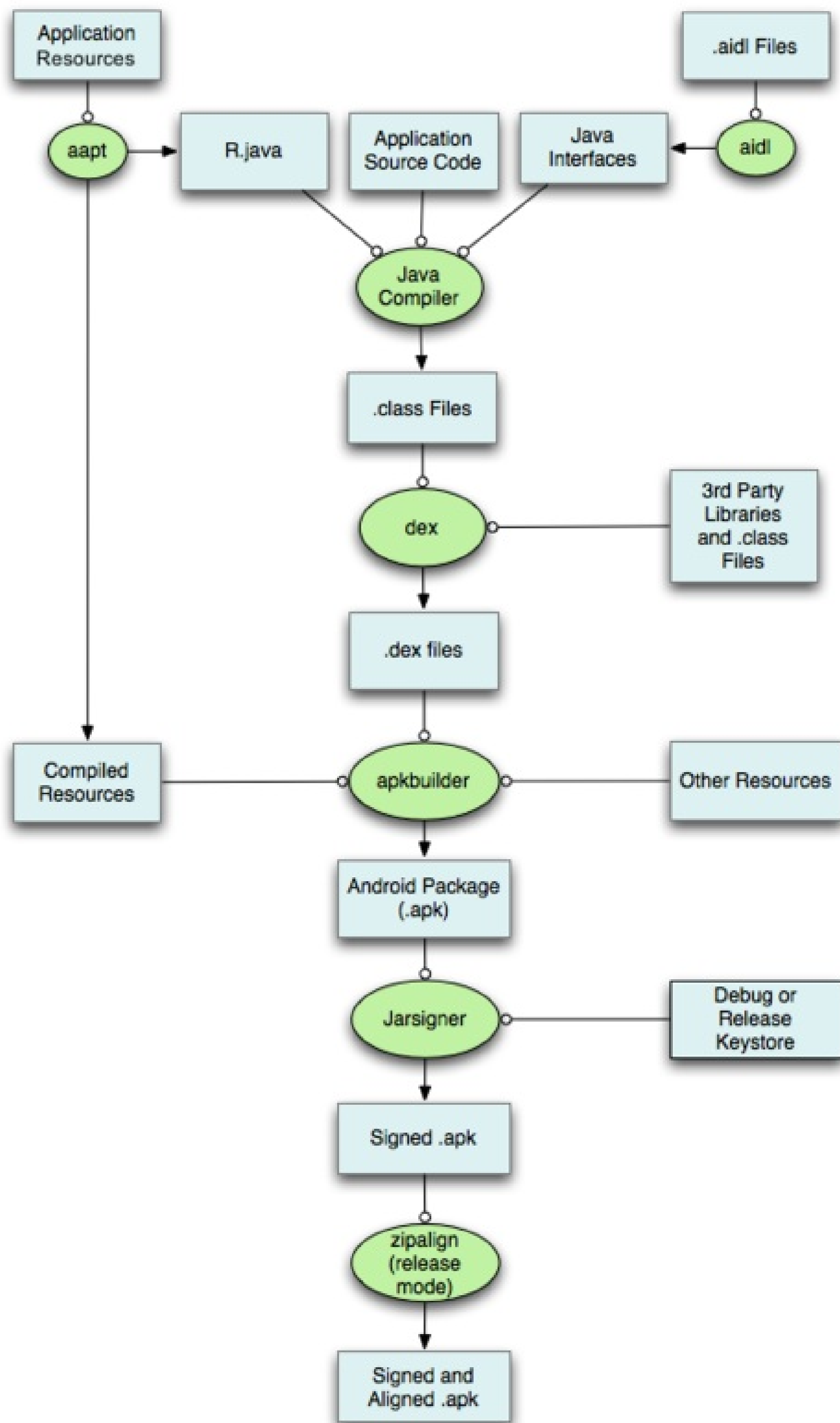
Java虚拟机基于栈架构，程序在运行时虚拟机需要频繁的从栈上读取或写入数据，这个过程需要更多的指令分派与内存访问次数，会耗费不少CPU时间，对于像手机设备资源有限的设备来说，这是相当大的一笔开销。**Dalvik**虚拟机基于寄存器架构。数据的访问通过寄存器间直接传递，这样的访问方式比基于栈方式要快很多。

四、Dalvik虚拟机的结构



一个应用首先经过DX工具将class文件转换成Dalvik虚拟机可以执行的dex文件，然后由类加载器加载原生类和Java类，接着由解释器根据指令集对Dalvik字节码进行解释、执行。最后，根据dvm_arch参数选择编译的目标机体系结构。

五、Android APK 编译打包流程



1.Java编译器对工程本身的java代码进行编译，这些java代码有三个来源：app的源代码，由资源文件生成的R文件(aapt工具)，以及有aidl文件生成的java接口文件(aidl工具)。产出为.class文件。

①用AAPT编译R.java文件

②编译AIDL的java文件

③把java文件编译成class文件

2..class文件和依赖的三方库文件通过dex工具生成Delvik虚拟机可执行的.dex文件，包含了所有的class信息，包括项目自身的class和依赖的class。产出为.dex文件。

3.apkbuilder工具将.dex文件和编译后的资源文件生成未经签名对齐的apk文件。这里编译后的资源文件包括两部分，一是由aapt编译产生的编译后的资源文件，二是依赖的三方库里的资源文件。产出为未经签名的.apk文件。

4.分别由Jarsigner和zipalign对apk文件进行签名和对齐，生成最终的apk文件。

总结为：编译-->DEX-->打包-->签名和对齐

六、ART虚拟机与Dalvik虚拟机的区别

什么是ART:

ART代表Android Runtime，其处理应用程序执行的方式完全不同于Dalvik，Dalvik是依靠一个Just-In-Time (JIT)编译器去解释字节码。开发者编译后的应用代码需要通过一个解释器在用户的设备上运行，这一机制并不高效，但让应用能更容易在不同硬件和架构上运行。ART则完全改变了这套做法，在应用安装时就预编译字节码到机器语言，这一机制叫Ahead-Of-Time (AOT）编译。在移除解释代码这一过程后，应用程序执行将更有效率，启动更快。

ART优点：

1. 系统性能的显著提升。
2. 应用启动更快、运行更快、体验更流畅、触感反馈更及时。
3. 更长的电池续航能力。
4. 支持更低的硬件。

ART缺点：

1. 更大的存储空间占用，可能会增加10%-20%。
2. 更长的应用安装时间。

ART虚拟机相对于Dalvik虚拟机的提升

预编译

在dalvik中，如同其他大多数JVM一样，都采用的是JIT来做及时翻译(动态翻译)，将dex或odex中并排的dalvik code(或者叫smali指令集)运行态翻译成native code去执行。JIT的引入使得dalvik提升了3~6倍的性能。

而在ART中，完全抛弃了dalvik的JIT，使用了AOT直接在安装时将其完全翻译成native code。这一技术的引入，使得虚拟机执行指令的速度又一重大提升。

垃圾回收机制

首先介绍下dalvik的GC的过程。主要有有四个过程:

1. 当gc被触发时候，其会去查找所有活动的对象，这个时候整个程序与虚拟机内部的所有线程就会挂起，这样目的是在较少的堆栈里找到所引用的对象；
    - 注意：这个回收动作和应用程序非并发；
2. gc对符合条件的对象进行标记；
3. gc对标记的对象进行回收；
4. 恢复所有线程的执行现场继续运行。

dalvik这么做的好处是，当**pause**了之后，**GC**势必是相当快速的。但是如果出现**GC**频繁并且内存吃紧势必会导致**UI**卡顿、掉帧、操作不流畅等。

后来ART改善了这种GC方式， 主要的改善点在将其非并发过程改成了部分并发，还有就是对内存的重新分配管理。

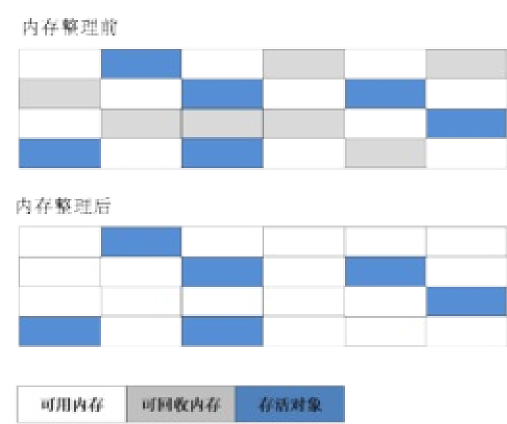
当ART GC发生时:

1. GC将会锁住Java堆，扫描并进行标记；
2. 标记完毕释放掉Java堆的锁，并且挂起所有线程；
3. GC对标记的对象进行回收；
4. 恢复所有线程的执行现场继续运行；
5. 重复**2-4**直到结束。

可以看出整个过程做到了部分并发使得时间缩短。据官方测试数据说GC效率提高2倍。

提高内存使用，减少碎片化

Dalvik内存管理特点是**:**内存碎片化严重，当然这也是Mark and Sweep算法带来的弊端。



可以看出每次GC后内存千疮百孔，本来连续分配的内存块变得碎片化严重，之后再分配进入的对象再进行内存寻址变得困难。

ART的解决：在ART中，它将Java分了一块空间命名为**Large-Object-Space**，这块内存空间的引入用来专门存放large object。同时ART又引入了moving collector的技术，即将不连续的物理内存块进行对齐。对齐了后内存碎片化就得到了很好的解决。Large-Object-Space的引入是因为moving collector对大块内存的位移时间成本太高。根官方统计，ART的内存利用率提高10倍了左右，大大提高了内存的利用率。

Android 中的 IPC 方式

一、使用 Intent

1. Activity，Service，Receiver 都支持在 Intent 中传递 Bundle 数据，而 Bundle 实现了 Parcelable 接口，可以在不同的进程间进行传输。
2. 在一个进程中启动了另一个进程的 Activity，Service 和 Receiver ，可以在 Bundle 中附加要传递的数据通过 Intent 发送出去。

二、使用文件共享

1. Windows 上，一个文件如果被加了排斥锁会导致其他线程无法对其进行访问，包括读和写；而 Android 系统基于 Linux ，使得其并发读取文件没有限制地进行，甚至允许两个线程同时对一个文件进行读写操作，尽管这样可能会出问题。
2. 可以在一个进程中序列化一个对象到文件系统中，在另一个进程中反序列化恢复这个对象（注意：并不是同一个对象，只是内容相同。）。
3. SharedPreferences 是个特例，系统对它的读 / 写有一定的缓存策略，即内存中会有一份 ShardPreferences 文件的缓存，系统对他的读 / 写就变得不可靠，当面对高并发的读写访问，SharedPreferences 有很多大的几率丢失数据。因此，IPC 不建议采用 SharedPreferences。

三、使用 Messenger

Messenger 是一种轻量级的 IPC 方案，它的底层实现是 AIDL ，可以在不同进程中传递 Message 对象，它一次只处理一个请求，在服务端不需要考虑线程同步的问题，服务端不存在并发执行的情形。

- 服务端进程：服务端创建一个 Service 来处理客户端请求，同时通过一个 Handler 对象来实例化一个 Messenger 对象，然后在 Service 的 onBind 中返回这个 Messenger 对象底层的 Binder 即可。

```
public class MessengerService extends Service {
```

```
    private static final String TAG = MessengerService.class.get
SimpleName();

    private class MessengerHandler extends Handler {

        /**
         * @param msg
         */
        @Override
        public void handleMessage(Message msg) {

            switch (msg.what) {
                case Constants.MSG_FROM_CLIENT:
                    Log.d(TAG, "receive msg from client: msg = ["
 + msg.getData().getString(Constants.MSG_KEY) + "]");
                    Toast.makeText(MessengerService.this, "recei
ve msg from client: msg = [" + msg.getData().getString(Constants
.MSG_KEY) + "]", Toast.LENGTH_SHORT).show();
                    Messenger client = msg.replyTo;
                    Message replyMsg = Message.obtain(null, Cons
tants.MSG_FROM_SERVICE);
                    Bundle bundle = new Bundle();
                    bundle.putString(Constants.MSG_KEY, "我已经收
到你的消息，稍后回复你！");
                    replyMsg.setData(bundle);
                    try {
                        client.send(replyMsg);
                    } catch (RemoteException e) {
                        e.printStackTrace();
                    }
                    break;
                default:
                    super.handleMessage(msg);
            }
        }
    }

    private Messenger mMessenger = new Messenger(new MessengerHa
ndler());
```

```
    @Nullable
    @Override
    public IBinder onBind(Intent intent) {
        return mMessenger.getBinder();
    }
}
```

- 客户端进程：首先绑定服务端 Service ，绑定成功之后用服务端的 IBinder 对象创建一个 Messenger ，通过这个 Messenger 就可以向服务端发送消息了，消息类型是 Message 。如果需要服务端响应，则需要创建一个 Handler 并通过它来创建一个 Messenger（和服务端一样），并通过 Message 的 replyTo 参数传递给服务端。服务端通过 Message 的 replyTo 参数就可以回应客户端了。

```
public class MainActivity extends AppCompatActivity {
    private static final String TAG = MainActivity.class.getSimp
leName();
    private Messenger mGetReplyMessenger = new Messenger(new Mes
sageHandler());
    private Messenger mService;

    private class MessageHandler extends Handler {
        @Override
        public void handleMessage(Message msg) {
            switch (msg.what) {
                case Constants.MSG_FROM_SERVICE:
                    Log.d(TAG, "received msg form service: msg =
 [" + msg.getData().getString(Constants.MSG_KEY) + "]");
                    Toast.makeText(MainActivity.this, "received
msg form service: msg = [" + msg.getData().getString(Constants.M
SG_KEY) + "]", Toast.LENGTH_SHORT).show();
                    break;
                default:
                    super.handleMessage(msg);
            }
        }
    }
```

```
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

    }

    public void bindService(View v) {
        Intent mIntent = new Intent(this, MessengerService.class
);
        bindService(mIntent, mServiceConnection, Context.BIND_AU
TO_CREATE);
    }

    public void sendMessage(View v) {
        Message msg = Message.obtain(null,Constants.MSG_FROM_CLI
ENT);
        Bundle data = new Bundle();
        data.putString(Constants.MSG_KEY, "Hello! This is client
.");
        msg.setData(data);
        msg.replyTo = mGetReplyMessenger;
        try {
            mService.send(msg);
        } catch (RemoteException e) {
            e.printStackTrace();
        }

    }

    @Override
    protected void onDestroy() {
        unbindService(mServiceConnection);
        super.onDestroy();
    }

    private ServiceConnection mServiceConnection = new ServiceCo
```

```
nnection() {
        /**
         * @param name
         * @param service
         */
        @Override
        public void onServiceConnected(ComponentName name, IBind
er service) {
            mService = new Messenger(service);
            Message msg = Message.obtain(null,Constants.MSG_FROM
_CLIENT);
            Bundle data = new Bundle();
            data.putString(Constants.MSG_KEY, "Hello! This is cl
ient.");
            msg.setData(data);
            //
            msg.replyTo = mGetReplyMessenger;
            try {
                mService.send(msg);
            } catch (RemoteException e) {
                e.printStackTrace();
            }

        }

        /**
         * @param name
         */
        @Override
        public void onServiceDisconnected(ComponentName name) {


        }
    };
}
```

注意：客户端和服务端是通过拿到对方的 Messenger 来发送 Message 的。只不过客户端通过 bindService onServiceConnected 而服务端通过 message.replyTo 来获得对方的 Messenger 。Messenger 中有一个 Hanlder 以串行的方式处理队列中

的消息。不存在并发执行，因此我们不用考虑线程同步的问题。

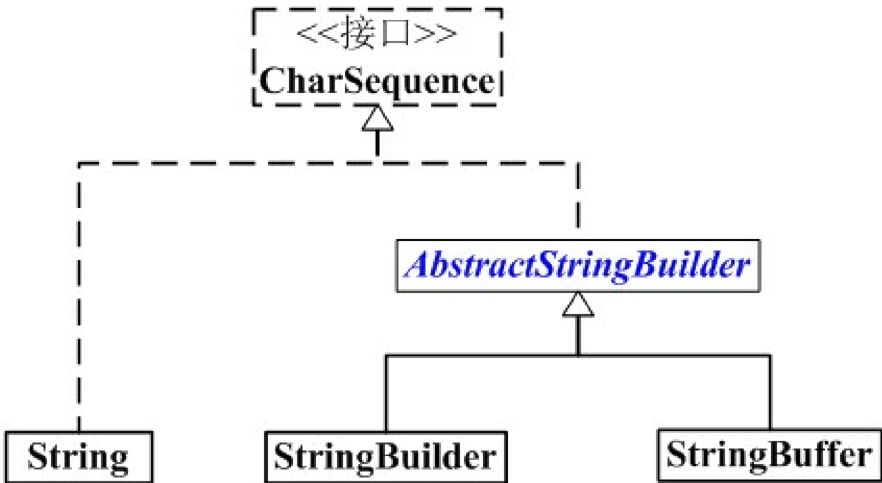


四、使用 AIDL

Messenger 是以串行的方式处理客户端发来的消息，如果大量消息同时发送到服务端，服务端只能一个一个处理，所以大量并发请求就不适合用 Messenger ，而且 Messenger 只适合传递消息，不能跨进程调用服务端的方法。AIDL 可以解决并发和跨进程调用方法的问题，要知道 Messenger 本质上也是 AIDL ，只不过系统做了封装方便上层的调用而已。

AIDL 文件支持的数据类型

- 基本数据类型；
- *String* 和 *CharSequence*



- *ArrayList* ，里面的元素必须能够被 AIDL 支持；
- *HashMap* ，里面的元素必须能够被 AIDL 支持；
- *Parcelable* ，实现 Parcelable 接口的对象； 注意：如果 **AIDL** 文件中用到了自定义的 **Parcelable** 对象，必须新建一个和它同名的 **AIDL** 文件。

- *AIDL* ，AIDL 接口本身也可以在 AIDL 文件中使用。

服务端

服务端创建一个 Service 用来监听客户端的连接请求，然后创建一个 AIDL 文件，将暴露给客户端的接口在这个 AIDL 文件中声明，最后在 Service 中实现这个 AIDL 接口即可。

客户端

绑定服务端的 Service ，绑定成功后，将服务端返回的 Binder 对象转成 AIDL 接口所属的类型，然后就可以调用 AIDL 中的方法了。客户端调用远程服务的方法，被调用的方法运行在服务端的 Binder 线程池中，同时客户端的线程会被挂起，如果服务端方法执行比较耗时，就会导致客户端线程长时间阻塞，导致 ANR 。客户端的 onServiceConnected 和 onServiceDisconnected 方法都在 UI 线程中。

服务端访问权限管理

- 使用 Permission 验证，在 manifest 中声明

```
<permission android:name="com.jc.ipc.ACCESS_BOOK_SERVICE"
    android:protectionLevel="normal"/>
<uses-permission android:name="com.jc.ipc.ACCESS_BOOK_SERVICE"/>
```

服务端 onBinder 方法中

```
public IBinder onBind(Intent intent) {
    //Permission 权限验证
    int check = checkCallingOrSelfPermission("com.jc.ipc.ACCESS_
BOOK_SERVICE");
    if (check == PackageManager.PERMISSION_DENIED) {
        return null;
    }

    return mBinder;
}
```

- Pid Uid 验证

详细代码：

```
// Book.aidl
package com.jc.ipc.aidl;

parcelable Book;
```

```
// IBookManager.aidl
package com.jc.ipc.aidl;

import com.jc.ipc.aidl.Book;
import com.jc.ipc.aidl.INewBookArrivedListener;

// AIDL 接口中只支持方法，不支持静态常量，区别于传统的接口
interface IBookManager {
    List<Book> getBookList();

    // AIDL 中除了基本数据类型，其他数据类型必须标上方向,in,out 或者 ino
ut
    // in 表示输入型参数
    // out 表示输出型参数
    // inout 表示输入输出型参数

    void addBook(in Book book);

    void registerListener(INewBookArrivedListener listener);
    void unregisterListener(INewBookArrivedListener listener);

}
```

```
// INewBookArrivedListener.aidl
package com.jc.ipc.aidl;
import com.jc.ipc.aidl.Book;

// 提醒客户端新书到来

interface INewBookArrivedListener {
    void onNewBookArrived(in Book newBook);
}
```

```
public class BookManagerActivity extends AppCompatActivity {
    private static final String TAG = BookManagerActivity.class.
getSimpleName();
    private static final int MSG_NEW_BOOK_ARRIVED = 0x10;
    private Button getBookListBtn,addBookBtn;
    private TextView displayTextView;
    private IBookManager bookManager;
    private Handler mHandler = new Handler(){
        @Override
        public void handleMessage(Message msg) {
            switch (msg.what) {
                case MSG_NEW_BOOK_ARRIVED:
                    Log.d(TAG, "handleMessage: new book arrived "
 + msg.obj);
                    Toast.makeText(BookManagerActivity.this, "ne
w book arrived " + msg.obj, Toast.LENGTH_SHORT).show();
                    break;
                default:
                    super.handleMessage(msg);
            }

        }
    };

    private ServiceConnection mServiceConn = new ServiceConnecti
on() {
        @Override
        public void onServiceConnected(ComponentName name, IBind
```

```
er service) {
            bookManager = IBookManager.Stub.asInterface(service)
;
            try {
                bookManager.registerListener(listener);
            } catch (RemoteException e) {
                e.printStackTrace();
            }

        }

        @Override
        public void onServiceDisconnected(ComponentName name) {

        }
    };

    private INewBookArrivedListener listener = new INewBookArriv
edListener.Stub() {
        @Override
        public void onNewBookArrived(Book newBook) throws Remote
Exception {
            mHandler.obtainMessage(MSG_NEW_BOOK_ARRIVED, newBook
).sendToTarget();

        }
    };

    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState)
{
        super.onCreate(savedInstanceState);
        setContentView(R.layout.book_manager);
        displayTextView = (TextView) findViewById(R.id.displayTe
xtView);
        Intent intent = new Intent(this, BookManagerService.clas
s);
        bindService(intent, mServiceConn, BIND_AUTO_CREATE);

    }
```

```
    public void getBookList(View view) {
        try {
            List<Book> list = bookManager.getBookList();
            Log.d(TAG, "getBookList: " + list.toString());
            displayTextView.setText(list.toString());
        } catch (RemoteException e) {
            e.printStackTrace();
        }

    }

    public void addBook(View view) {
        try {
            bookManager.addBook(new Book(3, "天龙八部"));
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }

    @Override
    protected void onDestroy() {
        if (bookManager != null && bookManager.asBinder().isBind
erAlive()) {
            Log.d(TAG, "unregister listener " + listener);
            try {
                bookManager.unregisterListener(listener);
            } catch (RemoteException e) {
                e.printStackTrace();
            }
        }
        unbindService(mServiceConn);
        super.onDestroy();
    }
}
```

```
public class BookManagerService extends Service {
```

```
    private static final String TAG = BookManagerService.class.g
etSimpleName();

    // CopyOnWriteArrayList 支持并发读写，实现自动线程同步，他不是继承
自 ArrayList
    private CopyOnWriteArrayList<Book> mBookList = new CopyOnWri
teArrayList<Book>();
    //对象是不能跨进程传输的，对象的跨进程传输本质都是反序列化的过程，Binde
r 会把客户端传递过来的对象重新转化生成一个新的对象
    //RemoteCallbackList 是系统专门提供的用于删除系统跨进程 listener
的接口，利用底层的 Binder 对象是同一个
    //RemoteCallbackList 会在客户端进程终止后，自动溢出客户端注册的 lis
tener ，内部自动实现了线程同步功能。
    private RemoteCallbackList<INewBookArrivedListener> mListene
rs = new RemoteCallbackList<>();
    private AtomicBoolean isServiceDestroied = new AtomicBoolean(
false);


    private Binder mBinder = new IBookManager.Stub() {

        @Override
        public List<Book> getBookList() throws RemoteException {
            return mBookList;
        }

        @Override
        public void addBook(Book book) throws RemoteException {
            Log.d(TAG, "addBook: " + book.toString());
            mBookList.add(book);

        }

        @Override
        public void registerListener(INewBookArrivedListener lis
tener) throws RemoteException {
            mListeners.register(listener);
        }

        @Override
```

```
        public void unregisterListener(INewBookArrivedListener l
istener) throws RemoteException {
            mListeners.unregister(listener);
        }
    };

    @Override
    public void onCreate() {
        super.onCreate();
        mBookList.add(new Book(1, "老人与海"));
        mBookList.add(new Book(2, "哈姆雷特"));
        new Thread(new ServiceWorker()).start();
    }

    private void onNewBookArrived(Book book) throws RemoteExcept
ion {
        mBookList.add(book);

        int count = mListeners.beginBroadcast();

        for (int i = 0; i < count; i++) {
            INewBookArrivedListener listener = mListeners.getBro
adcastItem(i);
            if (listener != null) {
                listener.onNewBookArrived(book);
            }
        }

        mListeners.finishBroadcast();

    }

    private class ServiceWorker implements Runnable {
        @Override
        public void run() {
            while (!isServiceDestroied.get()) {
                try {
                    Thread.sleep(5000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
```

```
                }
                int bookId = mBookList.size() +1;
                Book newBook = new Book(bookId, "new book # " +
bookId);
                try {
                    onNewBookArrived(newBook);
                } catch (RemoteException e) {
                    e.printStackTrace();
                }
            }

        }
    }


    @Nullable
    @Override
    public IBinder onBind(Intent intent) {
        //Permission 权限验证
        int check = checkCallingOrSelfPermission("com.jc.ipc.ACC
ESS_BOOK_SERVICE");
        if (check == PackageManager.PERMISSION_DENIED) {
            return null;
        }

        return mBinder;
    }

    @Override
    public void onDestroy() {
        isServiceDestroied.set(true);
        super.onDestroy();
    }
}
```

五、使用 ContentProvider

用于不同应用间数据共享，和 Messenger 底层实现同样是 Binder 和 AIDL，系统做了封装，使用简单。 系统预置了许多 ContentProvider ，如通讯录、日程表，需要跨进程访问。 使用方法：继承 ContentProvider 类实现 6 个抽象方法，这六个方法均运行在 ContentProvider 进程中，除 onCreate 运行在主线程里，其他五个方法均由外界回调运行在 Binder 线程池中。

ContentProvider 的底层数据，可以是 SQLite 数据库，可以是文件，也可以是内存中的数据。

详见代码：

```
public class BookProvider extends ContentProvider {
    private static final String TAG = "BookProvider";
    public static final String AUTHORITY = "com.jc.ipc.Book.Prov
ider";

    public static final Uri BOOK_CONTENT_URI = Uri.parse("conten
t://" + AUTHORITY + "/book");
    public static final Uri USER_CONTENT_URI = Uri.parse("conten
t://" + AUTHORITY + "/user");

    public static final int BOOK_URI_CODE = 0;
    public static final int USER_URI_CODE = 1;
    private static final UriMatcher sUriMatcher = new UriMatcher
(UriMatcher.NO_MATCH);

    static {
        sUriMatcher.addURI(AUTHORITY, "book", BOOK_URI_CODE);
        sUriMatcher.addURI(AUTHORITY, "user", USER_URI_CODE);
    }

    private Context mContext;
    private SQLiteDatabase mDB;

    @Override
    public boolean onCreate() {
        mContext = getContext();
        initProviderData();

        return true;
```

```
    }

    private void initProviderData() {
        //不建议在 UI 线程中执行耗时操作
        mDB = new DBOpenHelper(mContext).getWritableDatabase();
        mDB.execSQL("delete from " + DBOpenHelper.BOOK_TABLE_NAM
E);
        mDB.execSQL("delete from " + DBOpenHelper.USER_TABLE_NAM
E);
        mDB.execSQL("insert into book values(3,'Android');");
        mDB.execSQL("insert into book values(4,'iOS');");
        mDB.execSQL("insert into book values(5,'Html5');");
        mDB.execSQL("insert into user values(1,'haohao',1);");
        mDB.execSQL("insert into user values(2,'nannan',0);");

    }

    @Nullable
    @Override
    public Cursor query(Uri uri, String[] projection, String sel
ection, String[] selectionArgs, String sortOrder) {
        Log.d(TAG, "query, current thread"+ Thread.currentThread
());
        String table = getTableName(uri);
        if (table == null) {
            throw new IllegalArgumentException("Unsupported URI"
 + uri);
        }

        return mDB.query(table, projection, selection, selection
Args, null, null, sortOrder, null);
    }

    @Nullable
    @Override
    public String getType(Uri uri) {
        Log.d(TAG, "getType");
        return null;
    }
```

```
    @Nullable
    @Override
    public Uri insert(Uri uri, ContentValues values) {
        Log.d(TAG, "insert");
        String table = getTableName(uri);
        if (table == null) {
            throw new IllegalArgumentException("Unsupported URI"
 + uri);
        }
        mDB.insert(table, null, values);
        // 通知外界 ContentProvider 中的数据发生变化
        mContext.getContentResolver().notifyChange(uri, null);
        return uri;
    }

    @Override
    public int delete(Uri uri, String selection, String[] select
ionArgs) {
        Log.d(TAG, "delete");
        String table = getTableName(uri);
        if (table == null) {
            throw new IllegalArgumentException("Unsupported URI"
 + uri);
        }
        int count = mDB.delete(table, selection, selectionArgs);
        if (count > 0) {
            mContext.getContentResolver().notifyChange(uri, null
);
        }

        return count;
    }

    @Override
    public int update(Uri uri, ContentValues values, String sele
ction, String[] selectionArgs) {
        Log.d(TAG, "update");
        String table = getTableName(uri);
        if (table == null) {
            throw new IllegalArgumentException("Unsupported URI"
```

```
 + uri);
        }
        int row = mDB.update(table, values, selection, selection
Args);
        if (row > 0) {
            getContext().getContentResolver().notifyChange(uri,
null);
        }
        return row;
    }

    private String getTableName(Uri uri) {
        String tableName = null;
        switch (sUriMatcher.match(uri)) {
            case BOOK_URI_CODE:
                tableName = DBOpenHelper.BOOK_TABLE_NAME;
                break;
            case USER_URI_CODE:
                tableName = DBOpenHelper.USER_TABLE_NAME;
                break;
            default:
                break;
        }

        return tableName;

    }
}
```

```
public class DBOpenHelper extends SQLiteOpenHelper {

    private static final String DB_NAME = "book_provider.db";
    public static final String BOOK_TABLE_NAME = "book";
    public static final String USER_TABLE_NAME = "user";

    private static final int DB_VERSION = 1;

    private String CREATE_BOOK_TABLE = "CREATE TABLE IF NOT EXIS
TS "
            + BOOK_TABLE_NAME + "(_id INTEGER PRIMARY KEY," + "n
ame TEXT)";

    private String CREATE_USER_TABLE = "CREATE TABLE IF NOT EXIS
TS "
            + USER_TABLE_NAME + "(_id INTEGER PRIMARY KEY," + "n
ame TEXT,"
            + "sex INT)";



    public DBOpenHelper(Context context) {
        super(context, DB_NAME, null, DB_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        db.execSQL(CREATE_BOOK_TABLE);
        db.execSQL(CREATE_USER_TABLE);

    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int
 newVersion) {

    }
}
```

```
public class ProviderActivity extends AppCompatActivity {
    private static final String TAG = ProviderActivity.class.get
SimpleName();
    private TextView displayTextView;
    private Handler mHandler;

    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState)
{
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_provider);
        displayTextView = (TextView) findViewById(R.id.displayTe
xtView);
        mHandler = new Handler();

        getContentResolver().registerContentObserver(BookProvide
r.BOOK_CONTENT_URI, true, new ContentObserver(mHandler) {
            @Override
            public boolean deliverSelfNotifications() {
                return super.deliverSelfNotifications();
            }

            @Override
            public void onChange(boolean selfChange) {
                super.onChange(selfChange);
            }

            @Override
            public void onChange(boolean selfChange, Uri uri) {
                Toast.makeText(ProviderActivity.this, uri.toStri
ng(), Toast.LENGTH_SHORT).show();
                super.onChange(selfChange, uri);
            }
        });




    }
```

```
    public void insert(View v) {
        ContentValues values = new ContentValues();
        values.put("_id",1123);
        values.put("name", "三国演义");
        getContentResolver().insert(BookProvider.BOOK_CONTENT_UR
I, values);

    }
    public void delete(View v) {
        getContentResolver().delete(BookProvider.BOOK_CONTENT_UR
I, "_id = 4", null);


    }
    public void update(View v) {
        ContentValues values = new ContentValues();
        values.put("_id",1123);
        values.put("name", "三国演义新版");
        getContentResolver().update(BookProvider.BOOK_CONTENT_UR
I, values , "_id = 1123", null);


    }
    public void query(View v) {
        Cursor bookCursor = getContentResolver().query(BookProvi
der.BOOK_CONTENT_URI, new String[]{"_id", "name"}, null, null, n
ull);
        StringBuilder sb = new StringBuilder();
        while (bookCursor.moveToNext()) {
            Book book = new Book(bookCursor.getInt(0),bookCursor
.getString(1));
            sb.append(book.toString()).append("\n");
        }
        sb.append("--------------------------------").append("\n"
);
        bookCursor.close();

        Cursor userCursor = getContentResolver().query(BookProvi
der.USER_CONTENT_URI, new String[]{"_id", "name", "sex"}, null,
null, null);
```

```
        while (userCursor.moveToNext()) {
            sb.append(userCursor.getInt(0))
                    .append(userCursor.getString(1)).append(" ,"
)
                    .append(userCursor.getInt(2)).append(" ,")
                    .append("\n");
        }
        sb.append("--------------------------------");
        userCursor.close();
        displayTextView.setText(sb.toString());
    }
}
```

六、使用 Socket

> Socket起源于 Unix，而 Unix 基本哲学之一就是“一切皆文件”，都可以用“打开 open –读写 write/read –关闭 close ”模式来操作。Socket 就是该模式的一个实现，网络的 Socket 数据传输是一种特殊的 I/O，Socket 也是一种文件描述符。Socket 也具有一个类似于打开文件的函数调用： Socket()，该函数返回一个整型的Socket 描述符，随后的连接建立、数据传输等操作都是通过该 Socket 实现的。
>
> 常用的 Socket 类型有两种：流式 Socket（SOCK_STREAM）和数据报式 Socket（SOCK_DGRAM）。流式是一种面向连接的 Socket，针对于面向连接的 TCP 服务应用；数据报式 Socket 是一种无连接的 Socket ，对应于无连接的 UDP 服务应用。

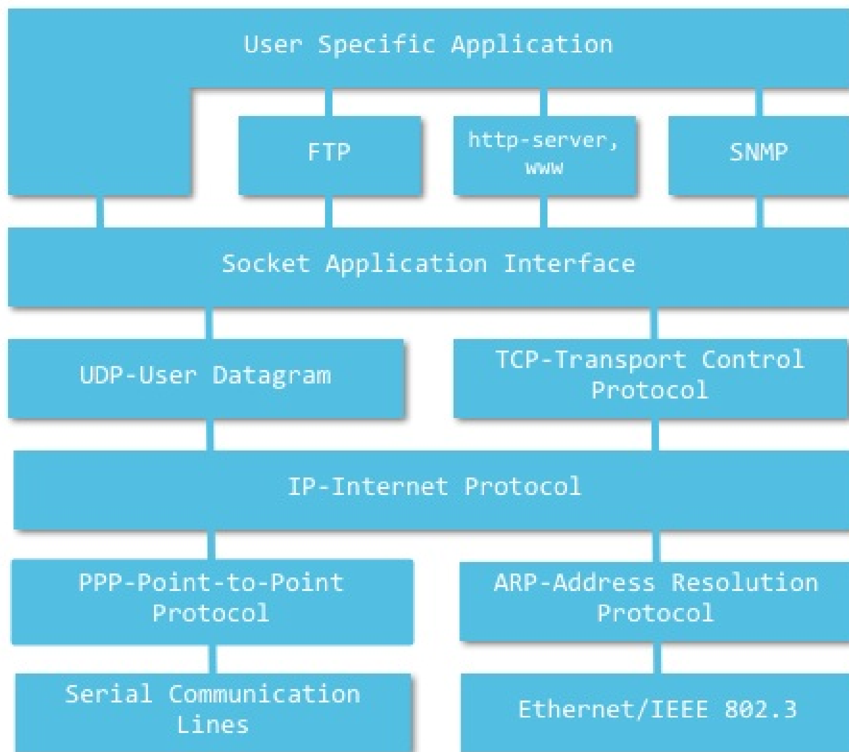
Socket 本身可以传输任意字节流。

谈到Socket，就必须要说一说 TCP/IP 五层网络模型：

- 应用层：规定应用程序的数据格式，主要的协议 HTTP，FTP，WebSocket，POP3 等；
- 传输层：建立“端口到端口” 的通信，主要的协议：TCP，UDP；
- 网络层：建立”主机到主机”的通信，主要的协议：IP，ARP ，IP 协议的主要作用：一个是为每一台计算机分配 IP 地址，另一个是确定哪些地址在同一子网；
- 数据链路层：确定电信号的分组方式，主要的协议：以太网协议；

- 物理层：负责电信号的传输。

Socket 是连接应用层与传输层之间接口（**API**）。



只实现 TCP Socket 。

Client 端代码：

```
public class TCPClientActivity extends AppCompatActivity impleme
nts View.OnClickListener{

    private static final String TAG = "TCPClientActivity";
    public static final int MSG_RECEIVED = 0x10;
    public static final int MSG_READY = 0x11;
    private EditText editText;
    private TextView textView;
    private PrintWriter mPrintWriter;
    private Socket mClientSocket;
    private Button sendBtn;
    private StringBuilder stringBuilder;
    private Handler mHandler = new Handler(){
        @Override
        public void handleMessage(Message msg) {
```

```
            switch (msg.what) {
                case MSG_READY:
                    sendBtn.setEnabled(true);
                    break;
                case MSG_RECEIVED:
                    stringBuilder.append(msg.obj).append("\n");
                    textView.setText(stringBuilder.toString());
                    break;
                default:
                    super.handleMessage(msg);
            }

    }
    };

    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState)
{
        super.onCreate(savedInstanceState);
        setContentView(R.layout.tcp_client_activity);
        editText = (EditText) findViewById(R.id.editText);
        textView = (TextView) findViewById(R.id.displayTextView)
;
        sendBtn = (Button) findViewById(R.id.sendBtn);
        sendBtn.setOnClickListener(this);
        sendBtn.setEnabled(false);
        stringBuilder = new StringBuilder();

        Intent intent = new Intent(TCPClientActivity.this, TCPSe
rverService.class);
        startService(intent);

        new Thread(){
            @Override
            public void run() {
                connectTcpServer();
            }
        }.start();
    }
```

```
    private String formatDateTime(long time) {
        return new SimpleDateFormat("(HH:mm:ss)").format(new Dat
e(time));
    }

    private void connectTcpServer() {
        Socket socket = null;
        while (socket == null) {
            try {
                socket = new Socket("localhost", 8888);
                mClientSocket = socket;
                mPrintWriter = new PrintWriter(new BufferedWrite
r(
                        new OutputStreamWriter(socket.getOutputS
tream())
                ), true);
                mHandler.sendEmptyMessage(MSG_READY);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }

        // receive message
        BufferedReader bufferedReader = null;
        try {
            bufferedReader = new BufferedReader(new InputStreamR
eader(socket.getInputStream()));
        } catch (IOException e) {
            e.printStackTrace();
        }
        while (!isFinishing()) {
            try {
                String msg = bufferedReader.readLine();
                if (msg != null) {
                    String time = formatDateTime(System.currentT
imeMillis());
                    String showedMsg = "server " + time + ":" +
msg
                            + "\n";
```

```
                    mHandler.obtainMessage(MSG_RECEIVED, showedM
sg).sendToTarget();
                }
            } catch (IOException e) {
                e.printStackTrace();
            }

        }
    }

    @Override
    public void onClick(View v) {
        if (mPrintWriter != null) {
            String msg = editText.getText().toString();
            mPrintWriter.println(msg);
            editText.setText("");
            String time = formatDateTime(System.currentTimeMilli
s());
            String showedMsg = "self " + time + ":" + msg + "\n"
;
            stringBuilder.append(showedMsg);

        }

    }

    @Override
    protected void onDestroy() {
        if (mClientSocket != null) {
            try {
                mClientSocket.shutdownInput();
                mClientSocket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        super.onDestroy();
    }
}
```

Server 端代码：

```
public class TCPServerService extends Service {
    private static final String TAG = "TCPServerService";
    private boolean isServiceDestroyed = false;
    private String[] mMessages = new String[]{
            "Hello! Body!",
            "用户不在线！请稍后再联系！",
            "请问你叫什么名字呀？",
            "厉害了，我的哥！",
            "Google 不需要科学上网是真的吗？",
            "扎心了，老铁！！！"
    };


    @Override
    public void onCreate() {
        new Thread(new TCPServer()).start();
        super.onCreate();
    }

    @Override
    public void onDestroy() {
        isServiceDestroyed = true;
        super.onDestroy();
    }

    @Nullable
    @Override
    public IBinder onBind(Intent intent) {
        return null;
    }

    private class TCPServer implements Runnable {

        @Override
        public void run() {
            ServerSocket serverSocket = null;
            try {
                serverSocket = new ServerSocket(8888);
```

```
            } catch (IOException e) {
                e.printStackTrace();
                return;
            }
            while (!isServiceDestroyed) {
                // receive request from client
                try {
                    final Socket client = serverSocket.accept();
                    Log.d(TAG, "=============== accept =========
=========");
                    new Thread(){
                        @Override
                        public void run() {
                            try {
                                responseClient(client);
                            } catch (IOException e) {
                                e.printStackTrace();
                            }
                        }
                    }.start();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }

        }
    }


    private void responseClient(Socket client) throws IOExceptio
n {
        //receive message
        BufferedReader in = new BufferedReader(
                new InputStreamReader(client.getInputStream()));
        //send message
        PrintWriter out = new PrintWriter(
                new BufferedWriter(
                        new OutputStreamWriter(
                                client.getOutputStream())),true)
;
```

```
        out.println("欢迎来到聊天室！");

        while (!isServiceDestroyed) {
            String str = in.readLine();
            Log.d(TAG, "message from client: " + str);
            if (str == null) {
                return;
            }
            Random random = new Random();
            int index = random.nextInt(mMessages.length);
            String msg = mMessages[index];
            out.println(msg);
            Log.d(TAG, "send Message: " + msg);
        }
        out.close();
        in.close();
        client.close();

    }
}
```

演示：



UDP Socket 可以自己尝试着实现。

一、为什么Bitmap需要高效加载？

现在的高清大图，动辄就要好几M，而Android对单个应用所施加的内存限制，只有小几十M，如16M，这导致加载Bitmap的时候很容易出现内存溢出。如下异常信息，便是在开发中经常需要的：

> java.lang.OutofMemoryError:bitmap size exceeds VM budget

为了解决这个问题，就出现了Bitmap的高效加载策略。其实核心思想很简单。假设通过ImageView来显示图片，很多时候ImageView并没有原始图片的尺寸那么大，这个时候把整个图片加载进来后再设置给ImageView，显然是没有必要的，因为ImageView根本没办法显示原始图片。这时候就可以按一定的采样率来将图片缩小后再加载进来，这样图片既能在ImageView显示出来，又能降低内存占用从而在一定程度上避免OOM，提高了Bitmap加载时的性能。

二、Bitmap高效加载的具体方式

1.加载Bitmap的方式

Bitmap在Android中指的是一张图片。通过BitmapFactory类提供的四类方法：decodeFile,decodeResource,decodeStream和decodeByteArray,分别从文件系统，资源，输入流和字节数组中加载出一个Bitmap对象，其中decodeFile,decodeResource又间接调用了decodeStream方法，这四类方法最终是在Android的底层实现的，对应着BitmapFactory类的几个native方法。

2.BitmapFactory.Options的参数

①inSampleSize参数

上述四类方法都支持BitmapFactory.Options参数，而Bitmap的按一定采样率进行缩放就是通过BitmapFactory.Options参数实现的，主要用到了inSampleSize参数，即采样率。通过对inSampleSize的设置，对图片的像素的高和宽进行缩放。

当inSampleSize=1，即采样后的图片大小为图片的原始大小。小于1，也按照1来计算。 当inSampleSize>1，即采样后的图片将会缩小，缩放比例为1/(inSampleSize的二次方)。

例如：一张1024 ×1024像素的图片，采用ARGB8888格式存储，那么内存大小1024×1024×4=4M。如果inSampleSize=2，那么采样后的图片内存大小：512×512×4=1M。

注意：官方文档支出，**inSampleSize**的取值应该总是**2**的指数，如**1**，**2**，**4**，**8**等。如果外界传入的**inSampleSize**的值不为**2**的指数，那么系统会向下取整并选择一个最接近**2**的指数来代替。比如**3**，系统会选择**2**来代替。当时经验证明并非在所有**Android**版本上都成立。

关于**inSampleSize**取值的注意事项： 通常是根据图片宽高实际的大小/需要的宽高大小，分别计算出宽和高的缩放比。但应该取其中最小的缩放比，避免缩放图片太小，到达指定控件中不能铺满，需要拉伸从而导致模糊。

例如：ImageView的大小是100×100像素，而图片的原始大小为200×300，那么宽的缩放比是2，高的缩放比是3。如果最终inSampleSize=2，那么缩放后的图片大小100×150，仍然合适ImageView。如果inSampleSize=3，那么缩放后的图片大小小于ImageView所期望的大小，这样图片就会被拉伸而导致模糊。

②inJustDecodeBounds参数

我们需要获取加载的图片的宽高信息，然后交给inSampleSize参数选择缩放比缩放。那么如何能先不加载图片却能获得图片的宽高信息，通过inJustDecodeBounds=true，然后加载图片就可以实现只解析图片的宽高信息，并不会真正的加载图片，所以这个操作是轻量级的。当获取了宽高信息，计算出缩放比后，然后在将inJustDecodeBounds=false,再重新加载图片，就可以加载缩放后的图片。

注意：**BitmapFactory**获取的图片宽高信息和图片的位置以及程序运行的设备有关，比如同一张图片放在不同的**drawable**目录下或者程序运行在不同屏幕密度的设备上，都可能导致**BitmapFactory**获取到不同的结果，和**Android**的资源加载机制有关。

3.高效加载Bitmap的流程

①将BitmapFactory.Options的inJustDecodeBounds参数设为true并加载图片。

②从BitmapFactory.Options中取出图片的原始宽高信息，它们对应于outWidth和outHeight参数。

③根据采样率的规则并结合目标View的所需大小计算出采样率inSampleSize。

④将BitmapFactory.Options的inJustDecodeBounds参数设为false，然后重新加载图片。

三、Bitmap高效加载的代码实现

```
 public static Bitmap decodeSampledBitmapFromResource(Resources
res, int resId, int reqWidth, int reqHeight){
        BitmapFactory.Options options = new BitmapFactory.Option
s();
        options.inJustDecodeBounds = true;
        //加载图片
        BitmapFactory.decodeResource(res,resId,options);
        //计算缩放比
        options.inSampleSize = calculateInSampleSize(options,req
Height,reqWidth);
        //重新加载图片
        options.inJustDecodeBounds =false;
        return BitmapFactory.decodeResource(res,resId,options);
    }

    private static int calculateInSampleSize(BitmapFactory.Optio
ns options, int reqHeight, int reqWidth) {
        int height = options.outHeight;
        int width = options.outWidth;
        int inSampleSize = 1;
        if(height>reqHeight||width>reqWidth){
            int halfHeight = height/2;
            int halfWidth = width/2;
            //计算缩放比，是2的指数
            while((halfHeight/inSampleSize)>=reqHeight&&(halfWid
th/inSampleSize)>=reqWidth){
                inSampleSize*=2;
            }
        }


        return inSampleSize;
    }
```

这个时候就可以通过如下方式高效加载图片：

```
mImageView.setImageBitmap(decodeSampledBitmapFromResource(getRes
ources(),R.mipmap.ic_launcher,100,100);
```

除了BitmapFactory的decodeResource方法，其他方法也可以类似实现。

一、Android 动画分类

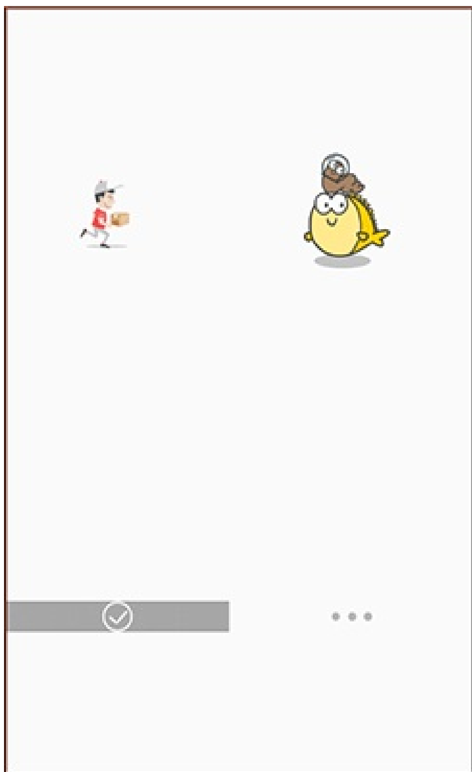
总的来说，Android动画可以分为两类，最初的传统动画和Android3.0 之后出现的属性动画；

传统动画又包括 帧动画（Frame Animation）和补间动画（Tweened Animation）。

二、传统动画

帧动画

帧动画是最容易实现的一种动画，这种动画更多的依赖于完善的UI资源，他的原理就是将一张张单独的图片连贯的进行播放，从而在视觉上产生一种动画的效果；有点类似于某些软件制作gif动画的方式。



如上图中的京东加载动画，代码要做的事情就是把一幅幅的图片按顺序显示，造成动画的视觉效果。

京东动画实现

```
<?xml version="1.0" encoding="utf-8"?>
<animation-list xmlns:android="http://schemas.android.com/apk/re
s/android">
    <item
        android:drawable="@drawable/a_0"
        android:duration="100" />
    <item
        android:drawable="@drawable/a_1"
        android:duration="100" />
    <item
        android:drawable="@drawable/a_2"
        android:duration="100" />
</animation-list>
```

```
protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_frame_animation);
        ImageView animationImg1 = (ImageView) findViewById(R.id.
animation1);
        animationImg1.setImageResource(R.drawable.frame_anim1);
        AnimationDrawable animationDrawable1 = (AnimationDrawabl
e) animationImg1.getDrawable();
        animationDrawable1.start();
    }
```

可以说，图片资源决定了这种方式可以实现怎样的动画

在有些代码中，我们还会看到*android：oneshot="false"*，这个*oneshot* 的含义就是动画执行一次（*true*）还是循环执行多次。

这里其他几个动画实现方式都是一样，无非就是图片资源的差异。

补间动画

补间动画又可以分为四种形式，分别是 **alpha**（淡入淡出），**translate**（位移），**scale**（缩放大小），**rotate**（旋转）。补间动画的实现，一般会采用xml 文件的形式；代码会更容易书写和阅读，同时也更容易复用。

XML 实现

首先，在res/anim/ 文件夹下定义如下的动画实现方式

alpha_anim.xml 动画实现

```
<?xml version="1.0" encoding="utf-8"?>
<alpha xmlns:android="http://schemas.android.com/apk/res/android"

    android:duration="1000"
    android:fromAlpha="1.0"
    android:interpolator="@android:anim/accelerate_decelerate_in
terpolator"
    android:toAlpha="0.0" />
```

scale.xml 动画实现

```
<?xml version="1.0" encoding="utf-8"?>
<scale xmlns:android="http://schemas.android.com/apk/res/android"

    android:duration="1000"
    android:fromXScale="0.0"
    android:fromYScale="0.0"
    android:pivotX="50%"
    android:pivotY="50%"
    android:toXScale="1.0"
    android:toYScale="1.0"/>
```

然后，在Activity中

```
Animation animation = AnimationUtils.loadAnimation(mContext, R.a
nim.alpha_anim);
img = (ImageView) findViewById(R.id.img);
img.startAnimation(animation);
```

这样就可以实现ImageView alpha 透明变化的动画效果。

也可以使用set 标签将多个动画组合（代码源自Android SDK API）

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android"
    android:interpolator="@[package:]anim/interpolator_resource"
    android:shareInterpolator=["true" | "false"] >
    <alpha
        android:fromAlpha="float"
        android:toAlpha="float" />
    <scale
        android:fromXScale="float"
        android:toXScale="float"
        android:fromYScale="float"
        android:toYScale="float"
        android:pivotX="float"
        android:pivotY="float" />
    <translate
        android:fromXDelta="float"
        android:toXDelta="float"
        android:fromYDelta="float"
        android:toYDelta="float" />
    <rotate
        android:fromDegrees="float"
        android:toDegrees="float"
        android:pivotX="float"
        android:pivotY="float" />
    <set>
        ...
    </set>
</set>
```

可以看到组合动画是可以嵌套使用的。

各个动画属性的含义结合动画自身的特点应该很好理解，就不一一阐述了；这里主要说一下**interpolator** 和 **pivot**。

> Interpolator 主要作用是可以控制动画的变化速率 ，就是动画进行的快慢节奏。

Android 系统已经为我们提供了一些Interpolator ，比如accelerate_decelerate_interpolator，accelerate_interpolator等。更多的interpolator 及其含义可以在Android SDK 中查看。同时这个Interpolator也是可以自定义的，这个后面还会提到。

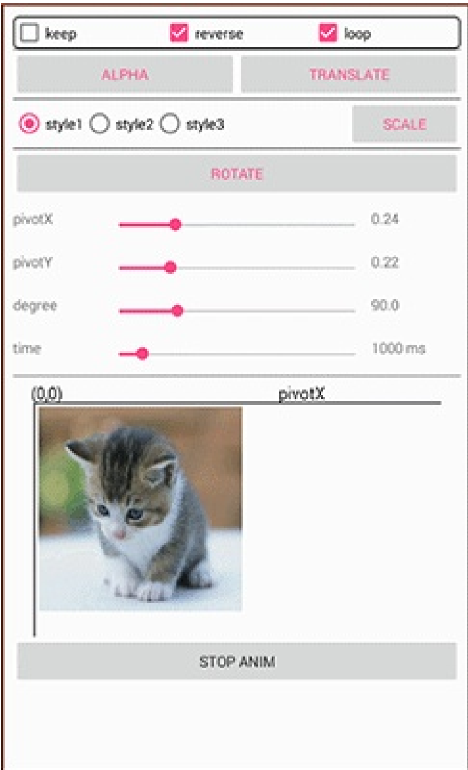
> pivot 决定了当前动画执行的参考位置

pivot 这个属性主要是在translate 和 scale 动画中，这两种动画都牵扯到view 的“物理位置“发生变化，所以需要一个参考点。而pivotX和pivotY就共同决定了这个点；它的值可以是float或者是百分比数值。

我们以pivotX为例，

pivotX取值	含义
10	距离动画所在view自身左边缘10像素
10%	距离动画所在view自身左边缘 的距离是整个view宽度的10%
10%p	距离动画所在view父控件左边缘的距离是整个view宽度的10%

pivotY 也是相同的原理，只不过变成的纵向的位置。如果还是不明白可以参考源码，在Tweened Animation中结合seekbar的滑动观察rotate的变化理解。



Java Code 实现

有时候，动画的属性值可能需要动态的调整，这个时候使用xml 就不合适了，需要使用java代码实现

```
private void RotateAnimation() {
        animation = new RotateAnimation(-deValue, deValue, Anima
tion.RELATIVE_TO_SELF,
                pxValue, Animation.RELATIVE_TO_SELF, pyValue);
        animation.setDuration(timeValue);

        if (keep.isChecked()) {
            animation.setFillAfter(true);
        } else {
            animation.setFillAfter(false);
        }
        if (loop.isChecked()) {
            animation.setRepeatCount(-1);
        } else {
            animation.setRepeatCount(0);
        }

        if (reverse.isChecked()) {
            animation.setRepeatMode(Animation.REVERSE);
        } else {
            animation.setRepeatMode(Animation.RESTART);
        }
        img.startAnimation(animation);
    }
```

这里**animation.setFillAfter**决定了动画在播放结束时是否保持最终的状态；**animation.setRepeatCount**和**animation.setRepeatMode** 决定了动画的重复次数及重复方式，具体细节可查看源码理解。

好了，传统动画的内容就说到这里了。

三、属性动画

属性动画，顾名思义它是对于对象属性的动画。因此，所有补间动画的内容，都可以通过属性动画实现。

属性动画入门

首先我们来看看如何用属性动画实现上面补间动画的效果

```
    private void RotateAnimation() {
        ObjectAnimator anim = ObjectAnimator.ofFloat(myView, "rotation", 0f, 360f);
        anim.setDuration(1000);
        anim.start();
    }

    private void AlpahAnimation() {
        ObjectAnimator anim = ObjectAnimator.ofFloat(myView, "alpha", 1.0f, 0.8f, 0.6f, 0.4f, 0.2f, 0.0f);
        anim.setRepeatCount(-1);
        anim.setRepeatMode(ObjectAnimator.REVERSE);
        anim.setDuration(2000);
        anim.start();
    }
```

这两个方法用属性动画的方式分别实现了旋转动画和淡入淡出动画，其中setDuration、setRepeatMode及setRepeatCount和补间动画中的概念是一样的。

可以看到，属性动画貌似强大了许多，实现很方便，同时动画可变化的值也有了更多的选择，动画所能呈现的细节也更多。

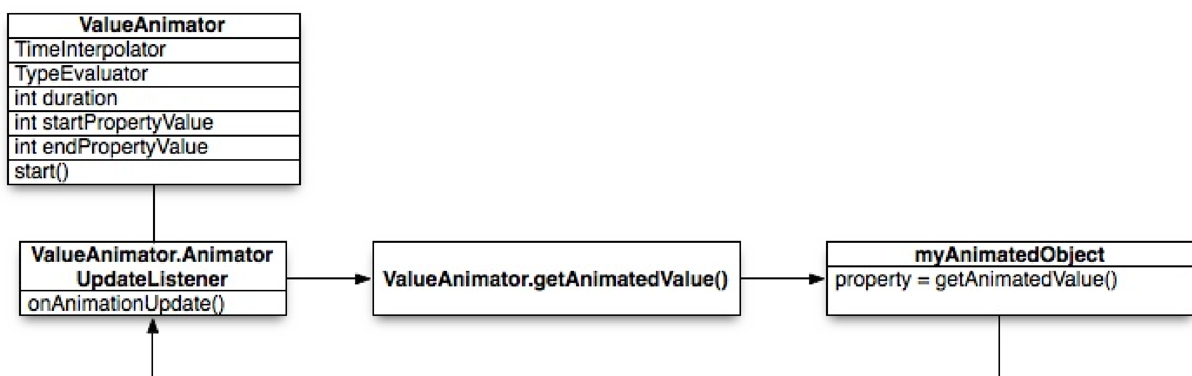
当然属性动画也是可以组合实现的

```
                ObjectAnimator alphaAnim = ObjectAnimator.ofFloa
t(myView, "alpha", 1.0f, 0.5f, 0.8f, 1.0f);
                ObjectAnimator scaleXAnim = ObjectAnimator.ofFlo
at(myView, "scaleX", 0.0f, 1.0f);
                ObjectAnimator scaleYAnim = ObjectAnimator.ofFlo
at(myView, "scaleY", 0.0f, 2.0f);
                ObjectAnimator rotateAnim = ObjectAnimator.ofFlo
at(myView, "rotation", 0, 360);
                ObjectAnimator transXAnim = ObjectAnimator.ofFlo
at(myView, "translationX", 100, 400);
                ObjectAnimator transYAnim = ObjectAnimator.ofFlo
at(myView, "tranlsationY", 100, 750);
                AnimatorSet set = new AnimatorSet();
                set.playTogether(alphaAnim, scaleXAnim, scaleYAn
im, rotateAnim, transXAnim, transYAnim);
//                set.playSequentially(alphaAnim, scaleXAnim, sc
aleYAnim, rotateAnim, transXAnim, transYAnim);
                set.setDuration(3000);
                set.start();
```

可以看到这些动画可以同时播放，或者是按序播放。

属性动画核心原理

在上面实现属性动画的时候，我们反复的使用到了ObjectAnimator 这个类，这个类继承自ValueAnimator，使用这个类可以对任意对象的任意属性进行动画操作。而ValueAnimator是整个属性动画机制当中最核心的一个类；这点从下面的图片也可以看出。



属性动画核心原理，此图来自于Android SDK API 文档。

> 属性动画的运行机制是通过不断地对值进行操作来实现的，而初始值和结束值之间的动画过渡就是由ValueAnimator这个类来负责计算的。它的内部使用一种时间循环的机制来计算值与值之间的动画过渡，我们只需要将初始值和结束值提供给ValueAnimator，并且告诉它动画所需运行的时长，那么ValueAnimator就会自动帮我们完成从初始值平滑地过渡到结束值这样的效果。除此之外，ValueAnimator还负责管理动画的播放次数、播放模式、以及对动画设置监听器等。

从上图我们可以了解到，通过duration、startPropertyValue和endPropertyValue 等值，我们就可以定义动画运行时长，初始值和结束值。然后通过start方法开始动画。 那么ValueAnimator 到底是怎样实现从初始值平滑过渡到结束值的呢？这个就是由TypeEvaluator 和TimeInterpolator 共同决定的。

具体来说，**TypeEvaluator** 决定了动画如何从初始值过渡到结束值。

TimeInterpolator 决定了动画从初始值过渡到结束值的节奏。

说的通俗一点，你每天早晨出门去公司上班，TypeEvaluator决定了你是坐公交、坐地铁还是骑车；而当你决定骑车后，TimeInterpolator决定了你一路上骑行的方式，你可以匀速的一路骑到公司，你也可以前半程骑得飞快，后半程骑得慢悠悠。

如果，还是不理解，那么就看下面的代码吧。首先看一下下面的这两个gif动画，一个小球在屏幕上以 y=sin(x) 的数学函数轨迹运行，同时小球的颜色和半径也发生着变化，可以发现，两幅图动画变化的节奏也是不一样的。



如果不考虑属性动画，这样的一个动画纯粹的使用*Canvas+Handler*的方式绘制也是有可能实现的。但是会复杂很多，而且加上各种线程，会带来很多意想不到的问题。

这里就通过自定义属性动画的方式看看这个动画是如何实现的。

属性动画自定义实现

这个动画最关键的三点就是 运动轨迹、小球半径及颜色的变化；我们就从这三个方面展开。最后我们在结合Interpolator说一下TimeInterpolator的意义。

用**TypeEvaluator** 确定运动轨迹

前面说了，TypeEvaluator决定了动画如何从初始值过渡到结束值。这个TypeEvaluator是个接口，我们可以实现这个接口。

```
public class PointSinEvaluator implements TypeEvaluator {

    @Override
    public Object evaluate(float fraction, Object startValue, Ob
ject endValue) {
        Point startPoint = (Point) startValue;
        Point endPoint = (Point) endValue;
        float x = startPoint.getX() + fraction * (endPoint.getX(
) - startPoint.getX());

        float y = (float) (Math.sin(x * Math.PI / 180) * 100) +
endPoint.getY() / 2;
        Point point = new Point(x, y);
        return point;
    }
}
```

PointSinEvaluator 继承了TypeEvaluator类，并实现了他唯一的方法evaluate；这个方法有三个参数，第一个参数fraction 代表当前动画完成的百分比，这个值是如何变化的后面还会提到；第二个和第三个参数代表动画的初始值和结束值。这里我们的逻辑很简单，x的值随着fraction 不断变化，并最终达到结束值；y的值就是当前x值所对应的sin(x) 值，然后用x 和 y 产生一个新的点（Point对象）返回。

这样我们就可以使用这个PointSinEvaluator 生成属性动画的实例了。

```
        Point startP = new Point(RADIUS, RADIUS);//初始值（起点）
        Point endP = new Point(getWidth() - RADIUS, getHeight()
- RADIUS);//结束值（终点）
        final ValueAnimator valueAnimator = ValueAnimator.ofObje
ct(new PointSinEvaluator(), startP, endP);
        valueAnimator.setRepeatCount(-1);
        valueAnimator.setRepeatMode(ValueAnimator.REVERSE);
        valueAnimator.addUpdateListener(new ValueAnimator.Animat
orUpdateListener() {
            @Override
            public void onAnimationUpdate(ValueAnimator animatio
n) {
                currentPoint = (Point) animation.getAnimatedValu
e();
                postInvalidate();
            }
        });
```

这样我们就完成了动画轨迹的定义，现在只要调用valueAnimator.start() 方法，就会绘制出一个正弦曲线的轨迹。

颜色及半径动画实现

之前我们说过，使用ObjectAnimator 可以对任意对象的任意属性进行动画操作，这句话是不太严谨的，这个任意属性还需要有get 和 set 方法。

```
public class PointAnimView extends View {

    /**
     * 实现关于color 的属性动画
     */
    private int color;
    private float radius = RADIUS;

    .....

}
```

这里在我们的自定义view中，定义了两个属性color 和 radius，并实现了他们各自的get set 方法，这样我们就可以使用属性动画的特点实现小球颜色变化的动画和半径变化的动画。

```
        ObjectAnimator animColor = ObjectAnimator.ofObject(this,
"color", new ArgbEvaluator(), Color.GREEN,
                Color.YELLOW, Color.BLUE, Color.WHITE, Color.RED
);
        animColor.setRepeatCount(-1);
        animColor.setRepeatMode(ValueAnimator.REVERSE);


        ValueAnimator animScale = ValueAnimator.ofFloat(20f, 80f
, 60f, 10f, 35f,55f,10f);
        animScale.setRepeatCount(-1);
        animScale.setRepeatMode(ValueAnimator.REVERSE);
        animScale.setDuration(5000);
        animScale.addUpdateListener(new ValueAnimator.AnimatorUp
dateListener() {
            @Override
            public void onAnimationUpdate(ValueAnimator animatio
n) {
                radius = (float) animation.getAnimatedValue();
            }
        });
```

这里，我们使用ObjectAnimator 实现对color 属性的值按照ArgbEvaluator 这个类的规律在给定的颜色值之间变化，这个ArgbEvaluator 和我们之前定义的PointSinEvaluator一样，都是决定动画如何从初始值过渡到结束值的，只不过这个类是系统自带的，我们直接拿来用就可以，他可以实现各种颜色间的自由过渡。

对radius 这个属性使用了ValueAnimator，使用了其ofFloat方法实现了一系列float值的变化；同时为其添加了动画变化的监听器，在属性值更新的过程中，我们可以将变化的结果赋给radius，这样就实现了半径动态的变化。

这里**radius** 也可以使用和**color**相同的方式，只需要把**ArgbEvaluator** 替换为**FloatEvaluator**，同时修改动画的变化值即可；使用添加监听器的方式，只是为了介绍监听器的使用方法而已

好了，到这里我们已经定义出了所有需要的动画，前面说过，属性动画也是可以组合使用的。因此，在动画启动的时候，同时播放这三个动画，就可以实现图中的效果了。

```
        animSet = new AnimatorSet();
        animSet.play(valueAnimator).with(animColor).with(animSca
le);
        animSet.setDuration(5000);
        animSet.setInterpolator(interpolatorType);
        animSet.start();
```

PointAnimView 源码

```
public class PointAnimView extends View {

    public static final float RADIUS = 20f;

    private Point currentPoint;

    private Paint mPaint;
    private Paint linePaint;

    private AnimatorSet animSet;
    private TimeInterpolator interpolatorType = new LinearInterp
olator();

    /**
     * 实现关于color 的属性动画
     */
    private int color;
    private float radius = RADIUS;

    public PointAnimView(Context context) {
        super(context);
        init();
    }


    public PointAnimView(Context context, AttributeSet attrs) {
```

```
        super(context, attrs);
        init();
    }

    public PointAnimView(Context context, AttributeSet attrs, int
defStyleAttr) {
        super(context, attrs, defStyleAttr);
        init();
    }


    public int getColor() {
        return color;
    }

    public void setColor(int color) {
        this.color = color;
        mPaint.setColor(this.color);
    }

    public float getRadius() {
        return radius;
    }

    public void setRadius(float radius) {
        this.radius = radius;
    }

    private void init() {
        mPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
        mPaint.setColor(Color.TRANSPARENT);

        linePaint = new Paint(Paint.ANTI_ALIAS_FLAG);
        linePaint.setColor(Color.BLACK);
        linePaint.setStrokeWidth(5);
    }

    @Override
    protected void onDraw(Canvas canvas) {
        if (currentPoint == null) {
```

```
            currentPoint = new Point(RADIUS, RADIUS);
            drawCircle(canvas);
//              StartAnimation();
        } else {
            drawCircle(canvas);
        }

        drawLine(canvas);
    }

    private void drawLine(Canvas canvas) {
        canvas.drawLine(10, getHeight() / 2, getWidth(), getHeig
ht() / 2, linePaint);
        canvas.drawLine(10, getHeight() / 2 - 150, 10, getHeight
() / 2 + 150, linePaint);
        canvas.drawPoint(currentPoint.getX(), currentPoint.getY(
), linePaint);

    }

    public void StartAnimation() {
        Point startP = new Point(RADIUS, RADIUS);
        Point endP = new Point(getWidth() - RADIUS, getHeight()
- RADIUS);
        final ValueAnimator valueAnimator = ValueAnimator.ofObje
ct(new PointSinEvaluator(), startP, endP);
        valueAnimator.setRepeatCount(-1);
        valueAnimator.setRepeatMode(ValueAnimator.REVERSE);
        valueAnimator.addUpdateListener(new ValueAnimator.Animat
orUpdateListener() {
            @Override
            public void onAnimationUpdate(ValueAnimator animatio
n) {
                currentPoint = (Point) animation.getAnimatedValu
e();
                postInvalidate();
            }
        });

//
```

```
        ObjectAnimator animColor = ObjectAnimator.ofObject(this,
"color", new ArgbEvaluator(), Color.GREEN,
                Color.YELLOW, Color.BLUE, Color.WHITE, Color.RED
);
        animColor.setRepeatCount(-1);
        animColor.setRepeatMode(ValueAnimator.REVERSE);


        ValueAnimator animScale = ValueAnimator.ofFloat(20f, 80f
, 60f, 10f, 35f,55f,10f);
        animScale.setRepeatCount(-1);
        animScale.setRepeatMode(ValueAnimator.REVERSE);
        animScale.setDuration(5000);
        animScale.addUpdateListener(new ValueAnimator.AnimatorUp
dateListener() {
            @Override
            public void onAnimationUpdate(ValueAnimator animatio
n) {
                radius = (float) animation.getAnimatedValue();
            }
        });


        animSet = new AnimatorSet();
        animSet.play(valueAnimator).with(animColor).with(animSca
le);
        animSet.setDuration(5000);
        animSet.setInterpolator(interpolatorType);
        animSet.start();

    }

    private void drawCircle(Canvas canvas) {
        float x = currentPoint.getX();
        float y = currentPoint.getY();
        canvas.drawCircle(x, y, radius, mPaint);
    }


    public void setInterpolatorType(int type ) {
```

```
        switch (type) {
            case 1:
                interpolatorType = new BounceInterpolator();
                break;
            case 2:
                interpolatorType = new AccelerateDecelerateInter
polator();
                break;
            case 3:
                interpolatorType = new DecelerateInterpolator();
                break;
            case 4:
                interpolatorType = new AnticipateInterpolator();
                break;
            case 5:
                interpolatorType = new LinearInterpolator();
                break;
            case 6:
                interpolatorType=new LinearOutSlowInInterpolator
();
                break;
            case 7:
                interpolatorType = new OvershootInterpolator();
            default:
                interpolatorType = new LinearInterpolator();
                break;
        }
    }


    @TargetApi(Build.VERSION_CODES.KITKAT)
    public void pauseAnimation() {
        if (animSet != null) {
            animSet.pause();
        }
    }


    public void stopAnimation() {
        if (animSet != null) {
```

```
                animSet.cancel();
                this.clearAnimation();
            }
        }
    }
```

TimeInterpolator 介绍

Interpolator的概念其实我们并不陌生，在补间动画中我们就使用到了。他就是用来控制动画快慢节奏的；而在属性动画中，TimeInterpolator 也是类似的作用；TimeInterpolator 继承自Interpolator。我们可以继承TimerInterpolator 以自己的方式控制动画变化的节奏，也可以使用Android 系统提供的Interpolator。

下面都是系统帮我们定义好的一些Interpolator，我们可以通过setInterpolator 设置不同的Interpolator。

Class/Interface	Description
`AccelerateDecelerateInterpolator`	An interpolator whose rate of change starts and ends slowly but accelerates through the middle.
`AccelerateInterpolator`	An interpolator whose rate of change starts out slowly and then accelerates.
`AnticipateInterpolator`	An interpolator whose change starts backward then flings forward.
`AnticipateOvershootInterpolator`	An interpolator whose change starts backward, flings forward and overshoots the target value, then finally goes back to the final value.
`BounceInterpolator`	An interpolator whose change bounces at the end.
`CycleInterpolator`	An interpolator whose animation repeats for a specified number of cycles.
`DecelerateInterpolator`	An interpolator whose rate of change starts out quickly and and then decelerates.
`LinearInterpolator`	An interpolator whose rate of change is constant.
`OvershootInterpolator`	An interpolator whose change flings forward and overshoots the last value then comes back.
`TimeInterpolator`	An interface that allows you to implement your own interpolator.

这里我们使用的Interpolator就决定了 前面我们提到的fraction。变化的节奏决定了动画所执行的百分比。不得不说，这么ValueAnimator的设计的确是很巧妙。

XML 属性动画

这里提一下，属性动画当然也可以使用xml文件的方式实现，但是属性动画的属性值一般会牵扯到对象具体的属性，更多是通过代码动态获取，所以xml文件的实现会有些不方便。

```
<set android:ordering="sequentially">
    <set>
        <objectAnimator
            android:propertyName="x"
            android:duration="500"
            android:valueTo="400"
            android:valueType="intType"/>
        <objectAnimator
            android:propertyName="y"
            android:duration="500"
            android:valueTo="300"
            android:valueType="intType"/>
    </set>
    <objectAnimator
        android:propertyName="alpha"
        android:duration="500"
        android:valueTo="1f"/>
</set>
```

使用方式：

```
AnimatorSet set = (AnimatorSet) AnimatorInflater.loadAnimator(my
Context,
    R.anim.property_animator);
set.setTarget(myObject);
set.start();
```

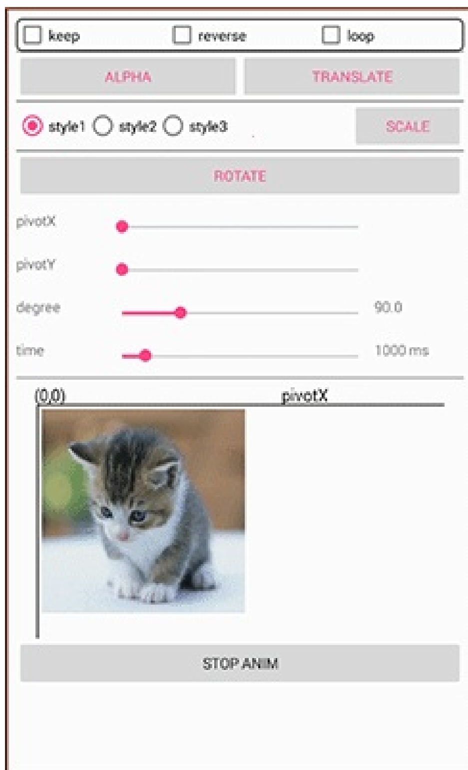
xml 文件中的标签也和属性动画的类相对应。

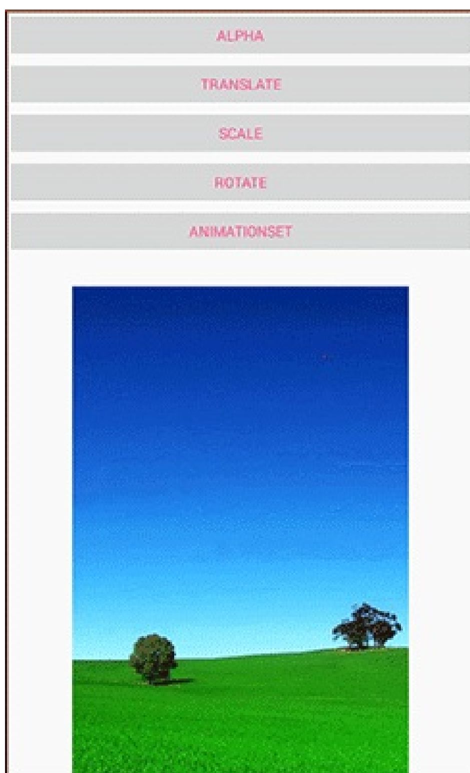
```
ValueAnimator --- <animator>
ObjectAnimator --- <objectAnimator>
AnimatorSet --- <set>
```

这些就是属性动画的核心内容。现在使用属性动画的特性自定义动画应该不是难事了。其余便签的含义，结合之前的内容应该不难理解了。

四、传统动画 VS 属性动画

相较于传统动画，属性动画有很多优势。那是否意味着属性动画可以完全替代传统动画呢。其实不然，两种动画都有各自的优势，属性动画如此强大，也不是没有缺点。





- 从上面两幅图比较可以发现，补间动画中，虽然使用translate将图片移动了，但是点击原来的位置，依旧可以发生点击事件，而属性动画却不是。因此我们可以确定，属性动画才是真正的实现了view的移动，补间动画对view的移动更像是在不同地方绘制了一个影子，实际的对象还是处于原来的地方。
- 当我们把动画的repeatCount设置为无限循环时，如果在Activity退出时没有及时将动画停止，属性动画会导致Activity无法释放而导致内存泄漏，而补间动画却没有问题。因此，使用属性动画时切记在Activity执行 onStop 方法时顺便将动画停止。（对这个怀疑的同学可以自己通过在动画的Update 回调方法打印日志的方式进行验证）。
- xml 文件实现的补间动画，复用率极高。在Activity切换，窗口弹出时等情景中有着很好的效果。
- 使用帧动画时需要注意，不要使用过多特别大的图，容易导致内存不足。

在安卓系统中：当系统内存不足时，Android系统将根据进程的优先级选择杀死一些不太重要的进程，优先级低的先杀死。进程优先级从高到低如下。

前台进程

- 处于正在与用户交互的activity
- 与前台activity绑定的service
- 调用了startForeground（）方法的service
- 正在执行oncreate（），onstart（），ondestroy方法的 service。
- 进程中包含正在执行onReceive（）方法的BroadcastReceiver。

系统中的前台进程并不会很多，而且一般前台进程都不会因为内存不足被杀死。特殊情况除外。当内存低到无法保证所有的前台进程同时运行时，才会选择杀死某个进程。

可视进程

- 为处于前台，但仍然可见的activity（例如：调用了onpause（）而还没调用onstop（）的activity）。典型情况是：运行activity时，弹出对话框（dialog等），此时的activity虽然不是前台activity，但是仍然可见。
- 可见activity绑定的service。（处于上诉情况下的activity所绑定的service）

可视进程一般也不会被系统杀死，除非为了保证前台进程的运行不得已而为之。

服务进程

- 已经启动的service

后台进程

- 不可见的activity（调用onstop（）之后的activity）

后台进程不会影响用户的体验，为了保证前台进程，可视进程，服务进程的运行，系统随时有可能杀死一个后台进程。当一个正确实现了生命周期的activity处于后台被杀死时，如果用户重新启动，会恢复之前的运行状态。

空进程

- 任何没有活动的进程

系统会杀死空进程，但这不会造成影响。空进程的存在无非为了一些缓存，以便于下次可以更快的启动。

Activity mActivity =new Activity()

作为Android开发者，不知道你有没有思考过这个问题，Activity可以new吗？Android的应用程序开发采用JAVA语言，Activity本质上也是一个对象，那上面的写法有什么问题呢？估计很多人说不清道不明。Android程序不像Java程序一样，随便创建一个类，写个main()方法就能运行，**Android**应用模型是基于组件的应用设计模式，组件的运行要有一个完整的**Android**工程环境，在这个环境下，Activity、Service等系统组件才能够正常工作，而这些组件并不能采用普通的Java对象创建方式，new一下就能创建实例了，而是要有它们各自的上下文环境，也就是我们这里讨论的Context。可以这样讲，Context是维持Android程序中各组件能够正常工作的一个核心功能类。

Context到底是什么?

Context的中文翻译为：语境; 上下文; 背景; 环境，在开发中我们经常说称之为“上下文”，那么这个“上下文”到底是指什么意思呢？在语文中，我们可以理解为语境，在程序中，我们可以理解为当前对象在程序中所处的一个环境，一个与系统交互的过程。比如微信聊天，此时的“环境”是指聊天的界面以及相关的数据请求与传输，Context在加载资源、启动Activity、获取系统服务、创建View等操作都要参与。

那Context到底是什么呢？一个Activity就是一个Context，一个Service也是一个Context。Android程序员把“场景”抽象为Context类，他们认为用户和操作系统的每一次交互都是一个场景，比如打电话、发短信，这些都是一个有界面的场景，还有一些没有界面的场景，比如后台运行的服务（Service）。一个应用程序可以认为是一个工作环境，用户在这个环境中会切换到不同的场景，这就像一个前台秘书，她可能需要接待客人，可能要打印文件，还可能要接听客户电话，而这些就称之为不同的场景，前台秘书可以称之为一个应用程序。

如何生动形象的理解Context?

上面的概念中采用了通俗的理解方式，将Context理解为“上下文”或者“场景”，如果你仍然觉得很抽象，不好理解。在这里我给出一个可能不是很恰当的比喻，希望有助于大家的理解：一个Android应用程序，可以理解为一部电影或者一部电视剧，Activity，Service，Broadcast Receiver，Content Provider这四大组件就好比是这部戏里的四个主角：胡歌，霍建华，诗诗，Baby。他们是由剧组（系统）一开始就定好了的，整部戏就是由这四位主演领衔担纲的，所以这四位主角并不是大街上随

随便便拉个人（new 一个对象）都能演的。有了演员当然也得有摄像机拍摄啊，他们必须通过镜头（Context）才能将戏传递给观众，这也就正对应说四大组件（四位主角）必须工作在Context环境下（摄像机镜头）。那Button，TextView，LinearLayout这些控件呢，就好比是这部戏里的配角或者说群众演员，他们显然没有这么重用，随便一个路人甲路人乙都能演（可以new一个对象），但是他们也必须要面对镜头（工作在Context环境下），所以 `Button mButton=new Button（Context）` 是可以的。虽然不很恰当，但还是很容易理解的，希望有帮助。

源码中的Context

```
/**
 * Interface to global information about an application environm
ent.  This is
 * an abstract class whose implementation is provided by
 * the Android system.  It
 * allows access to application-specific resources and classes,
as well as
 * up-calls for application-level operations such as launching a
ctivities,
 * broadcasting and receiving intents, etc.
 */
public abstract class Context {
    /**
     * File creation mode: the default mode, where the created f
ile can only
     * be accessed by the calling application (or all applicatio
ns sharing the
     * same user ID).
     * @see #MODE_WORLD_READABLE
     * @see #MODE_WORLD_WRITEABLE
     */
    public static final int MODE_PRIVATE = 0x0000;

    public static final int MODE_WORLD_WRITEABLE = 0x0002;

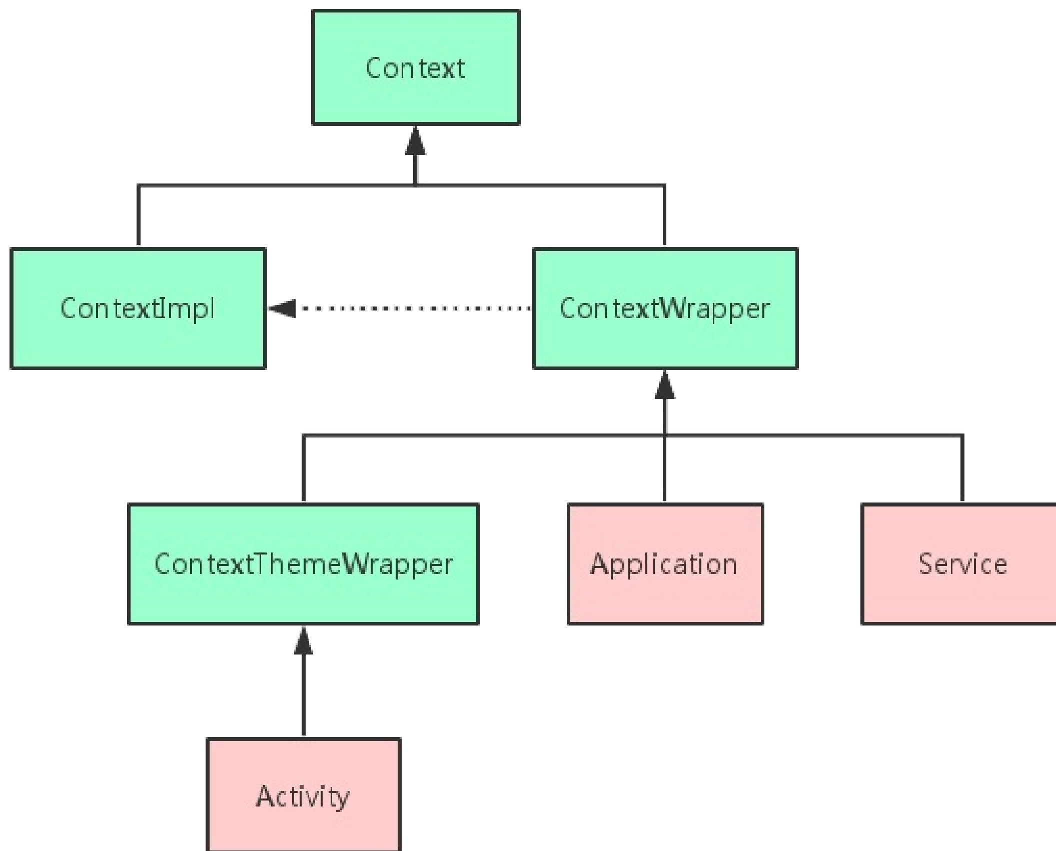
    public static final int MODE_APPEND = 0x8000;

    public static final int MODE_MULTI_PROCESS = 0x0004;

    .
    .
    .
    }
```

源码中的注释是这么来解释Context的：Context提供了关于应用环境全局信息的接口。它是一个抽象类，它的执行被**Android**系统所提供。它允许获取以应用为特征的资源和类型，是一个统领一些资源（应用程序环境变量等）的上下文。就是说，它描述一个应用程序环境的信息（即上下文）；是一个抽象类，Android提供了该

抽象类的具体实现类；通过它我们可以获取应用程序的资源和类（包括应用级别操作，如启动Activity，发广播，接受Intent等）。既然上面Context是一个抽象类，那么肯定有他的实现类咯，我们在Context的源码中通过IDE可以查看到他的子类最终可以得到如下关系图：



Context类本身是一个纯abstract类，它有两个具体的实现子类：ContextImpl和ContextWrapper。其中ContextWrapper类，如其名所言，这只是一个包装而已，ContextWrapper构造函数中必须包含一个真正的Context引用，同时ContextWrapper中提供了attachBaseContext（）用于给ContextWrapper对象中指定真正的Context对象，调用ContextWrapper的方法都会被转向其所包含的真正的Context对象。ContextThemeWrapper类，如其名所言，其内部包含了与主题（Theme）相关的接口，这里所说的主题就是指在AndroidManifest.xml中通过android：theme为Application元素或者Activity元素指定的主题。当然，只有Activity才需要主题，Service是不需要主题的，因为Service是没有界面的后台场景，所以Service直接继承于ContextWrapper，Application同理。而ContextImpl类则真正实现了Context中的所以函数，应用程序中所调用的各种Context类的方法，其实现均来自于该类。一句话总结：**Context的两个子类分工明确，其中ContextImpl是**

Context的具体实现类，**ContextWrapper**是**Context**的包装类。Activity，Application，Service虽都继承自ContextWrapper（Activity继承自ContextWrapper的子类ContextThemeWrapper），但它们初始化的过程中都会创建ContextImpl对象，由ContextImpl实现Context中的方法。

一个应用程序有几个Context？

其实这个问题本身并没有什么意义，关键还是在于对Context的理解，从上面的关系图我们已经可以得出答案了，在应用程序中Context的具体实现子类就是：Activity，Service，Application。那么 `Context数量=Activity数量+Service数量+1` 。当然如果你足够细心，可能会有疑问：我们常说四大组件，这里怎么只有Activity，Service持有Context，那Broadcast Receiver，Content Provider呢？**Broadcast Receiver**，**Content Provider**并不是**Context**的子类，他们所持有的**Context**都是其他地方传过去的，所以并不计入**Context**总数。上面的关系图也从另外一个侧面告诉我们Context类在整个Android系统中的地位是多么的崇高，因为很显然Activity，Service，Application都是其子类，其地位和作用不言而喻。

Context能干什么？

Context到底可以实现哪些功能呢？这个就实在是太多了，弹出Toast、启动Activity、启动Service、发送广播、操作数据库等等都需要用到Context。

```
TextView tv = new TextView(getContext());

ListAdapter adapter = new SimpleCursorAdapter(getApplicationCont
ext(), ...);

AudioManager am = (AudioManager) getContext().getSystemService(C
ontext.AUDIO_SERVICE);getApplicationContext().getSharedPreferenc
es(name, mode);

getApplicationContext().getContentResolver().query(uri, ...);

getContext().getResources().getDisplayMetrics().widthPixels * 5
/ 8;

getContext().startActivity(intent);

getContext().startService(intent);

getContext().sendBroadcast(intent);
```

Context作用域

虽然Context神通广大，但并不是随便拿到一个Context实例就可以为所欲为，它的使用还是有一些规则限制的。由于Context的具体实例是由ContextImpl类去实现的，因此在绝大多数场景下，Activity、Service和Application这三种类型的Context都是可以通用的。不过有几种场景比较特殊，比如启动Activity，还有弹出Dialog。出于安全原因的考虑，Android是不允许Activity或Dialog凭空出现的，一个Activity的启动必须要建立在另一个Activity的基础之上，也就是以此形成的返回栈。而Dialog则必须在一个Activity上面弹出（除非是System Alert类型的Dialog），因此在这种场景下，我们只能使用Activity类型的Context，否则将会出错。

Context作用域	Application	Activity	Service
Show a Dialog	No	YES	NO
Start an Activity	不推荐	YES	不推荐
Layout Inflation	不推荐	YES	不推荐
Start a Service	YES	YES	YES
Send a Broadcast	YES	YES	YES
Register Broadcast Receiver	YES	YES	YES
Load Resource Values	YES	YES	YES

从上图我们可以发现Activity所持有的Context的作用域最广，无所不能。因为Activity继承自ContextThemeWrapper，而Application和Service继承自ContextWrapper，很显然ContextThemeWrapper在ContextWrapper的基础上又做了一些操作使得Activity变得更强大，这里我就不再贴源码给大家分析了，有兴趣的童鞋可以自己查查源码。上图中的YES和NO我也不再做过多的解释了，这里我说一下上图中Application和Service所不推荐的两种使用情况。

1. 如果我们用ApplicationContext去启动一个LaunchMode为standard的Activity的时候会报错 `android.util.AndroidRuntimeException: Calling startActivity from outside of an Activity context requires the FLAG_ACTIVITY_NEW_TASK flag. Is this really what you want?`

   这是因为非Activity类型的Context并没有所谓的任务栈，所以待启动的Activity就找不到栈了。解决这个问题的方法就是为待启动的Activity指定FLAG_ACTIVITY_NEW_TASK标记位，这样启动的时候就为它创建一个新的任务栈，而此时Activity是以singleTask模式启动的。所有这种用Application启动Activity的方式不推荐使用，Service同Application。

2. 在Application和Service中去layout inflate也是合法的，但是会使用系统默认的主题样式，如果你自定义了某些样式可能不会被使用。所以这种方式也不推荐使用。

一句话总结：凡是跟UI相关的，都应该使用Activity做为Context来处理；其他的一些操作，Service,Activity,Application等实例都可以，当然了，注意Context引用的持有，防止内存泄漏。

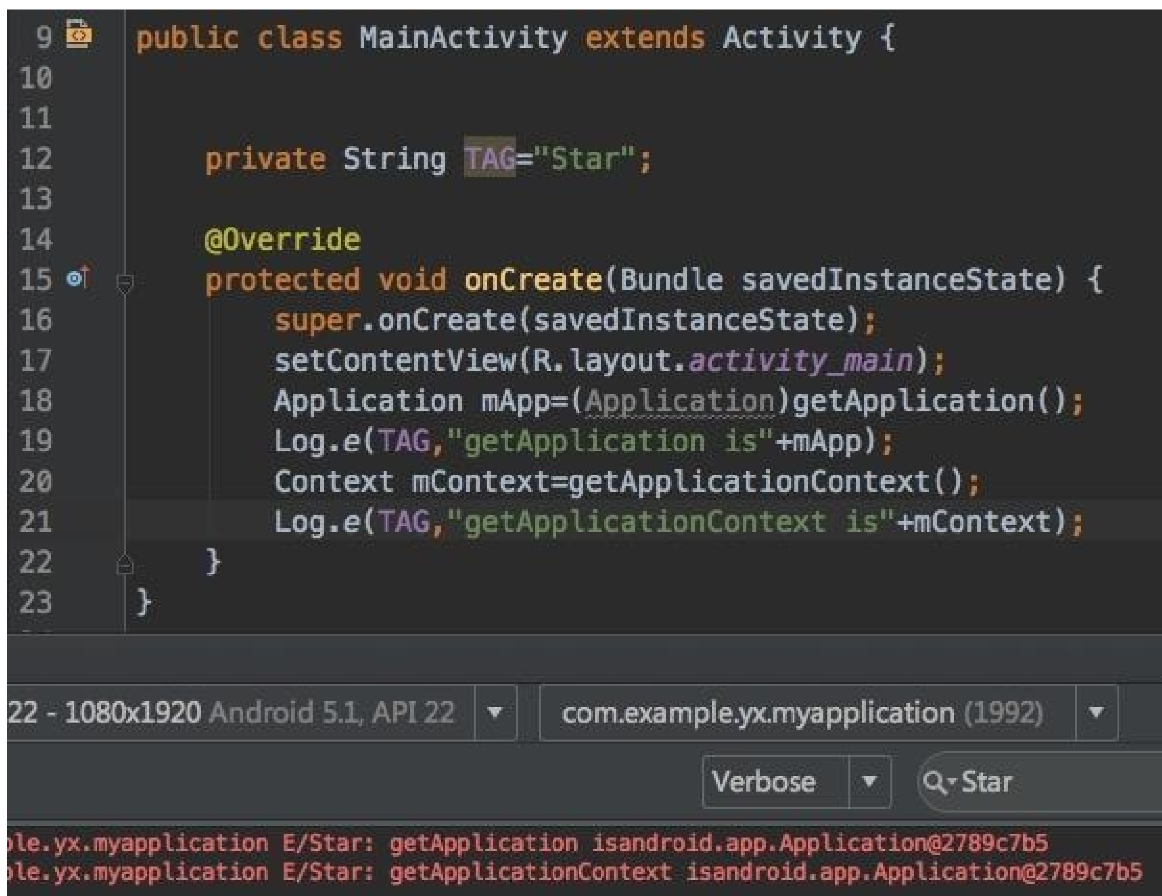
如何获取Context?

通常我们想要获取Context对象，主要有以下四种方法

1. View.getContext,返回当前View对象的Context对象，通常是当前正在展示的Activity对象。
2. Activity.getApplicationContext,获取当前Activity所在的(应用)进程的Context对象，通常我们使用Context对象时，要优先考虑这个全局的进程Context。
3. ContextWrapper.getBaseContext():用来获取一个ContextWrapper进行装饰之前的Context，可以使用这个方法，这个方法在实际开发中使用并不多，也不建议使用。
4. Activity.this 返回当前的Activity实例，如果是UI控件需要使用Activity作为Context对象，但是默认的Toast实际上使用ApplicationContext也可以。

getApplication()和getApplicationContext()

上面说到获取当前Application对象用getApplicationContext，不知道你有没有联想到getApplication()，这两个方法有什么区别？相信这个问题会难倒不少开发者。



程序是不会骗人的，我们通过上面的代码，打印得出两者的内存地址都是相同的，看来它们是同一个对象。其实这个结果也很好理解，因为前面已经说过了，Application本身就是一个Context，所以这里获取getApplicationContext()得到的结果就是Application本身的实例。那么问题来了，既然这两个方法得到的结果都是相同的，那么Android为什么要提供两个功能重复的方法呢？实际上这两个方法在作用域上有比较大的区别。getApplication()方法的语义性非常强，一看就知道是用来获取Application实例的，但是这个方法只有在**Activity**和**Service**中才能调用的到。那么也许在绝大多数情况下我们都是在Activity或者Service中使用Application的，但是如果在一些其它的场景，比如BroadcastReceiver中也想获得Application的实例，这时就可以借助getApplicationContext()方法了。

```
public class MyReceiver extends BroadcastReceiver{
  @Override
  public void onReceive(Contextcontext,Intentintent){
    Application myApp= (Application)context.getApplicationContex
t();
  }
}
```

Context引起的内存泄露

但Context并不能随便乱用，用的不好有可能会引起内存泄露的问题，下面就示例两种错误的引用方式。

错误的单例模式

```
public class Singleton {
    private static Singleton instance;
    private Context mContext;

    private Singleton(Context context) {
        this.mContext = context;
    }

    public static Singleton getInstance(Context context) {
        if (instance == null) {
            instance = new Singleton(context);
        }
        return instance;
    }
}
```

这是一个非线程安全的单例模式，instance作为静态对象，其生命周期要长于普通的对象，其中也包含Activity，假如Activity A去getInstance获得instance对象，传入this，常驻内存的Singleton保存了你传入的Activity A对象，并一直持有，即使Activity被销毁掉，但因为它的引用还存在于一个Singleton中，就不可能被GC掉，这样就导致了内存泄漏。

View持有Activity引用

```
public class MainActivity extends Activity {
    private static Drawable mDrawable;

    @Override
    protected void onCreate(Bundle saveInstanceState) {
        super.onCreate(saveInstanceState);
        setContentView(R.layout.activity_main);
        ImageView iv = new ImageView(this);
        mDrawable = getResources().getDrawable(R.drawable.ic_lau
ncher);
        iv.setImageDrawable(mDrawable);
    }
}
```

有一个静态的Drawable对象，当ImageView设置这个Drawable时，ImageView保存了mDrawable的引用，而ImageView传入的this是MainActivity的mContext，因为被static修饰的mDrawable是常驻内存的，MainActivity是它的间接引用，MainActivity被销毁时，也不能被GC掉，所以造成内存泄漏。

正确使用**Context**

一般Context造成的内存泄漏，几乎都是当Context销毁的时候，却因为被引用导致销毁失败，而Application的Context对象可以理解为随着进程存在的，所以我们总结出使用Context的正确姿势：

1. 当Application的Context能搞定的情况下，并且生命周期长的对象，优先使用Application的Context。
2. 不要让生命周期长于Activity的对象持有到Activity的引用。
3. 尽量不要在Activity中使用非静态内部类，因为非静态内部类会隐式持有外部类实例的引用，如果使用静态内部类，将外部实例引用作为弱引用持有。

一、前言

本部分内容是关于Android进阶的一些知识总结，涉及到的知识点比较杂，不过都是面试中几乎常问的知识点，也是加分的点。

关于这部分内容，可能需要有一些具体的项目实践。在面试的过程中，结合具体自身实践经历，才能更加深入透彻的描绘出来。

二、目录

一、原理

其实断点续传的原理很简单，从字面上理解，所谓断点续传就是从停止的地方重新下载。 断点：线程停止的位置。 续传：从停止的位置重新下载。

用代码解析就是： 断点 ： 当前线程已经下载完成的数据长度。 续传 ： 向服务器请求上次线程停止位置之后的数据。 原理知道了，功能实现起来也简单。每当线程停止时就把已下载的数据长度写入记录文件，当重新下载时，从记录文件读取已经下载了的长度。而这个长度就是所需要的断点。

续传的实现也简单，可以通过设置网络请求参数，请求服务器从指定的位置开始读取数据。 而要实现这两个功能只需要使用到httpURLconnection里面的setRequestProperty方法便可以实现.

```
public void setRequestProperty(String field, String newValue)
```

如下所示，便是向服务器请求500-1000之间的500个byte：

```
conn.setRequestProperty("Range", "bytes=" + 500 + "-" + 1000);
```

以上只是续传的一部分需求，当我们获取到下载数据时，还需要将数据写入文件，而普通发File对象并不提供从指定位置写入数据的功能，这个时候，就需要使用到RandomAccessFile来实现从指定位置给文件写入数据的功能。

```
public void seek(long offset)
```

如下所示，便是从文件的的第100个byte后开始写入数据。

```
raFile.seek(100);
```

而开始写入数据时还需要用到RandomAccessFile里面的另外一个方法

```
public void write(byte[] buffer, int byteOffset, int byteCount)
```

该方法的使用和OutputStream的write的使用一模一样...

以上便是断点续传的原理。

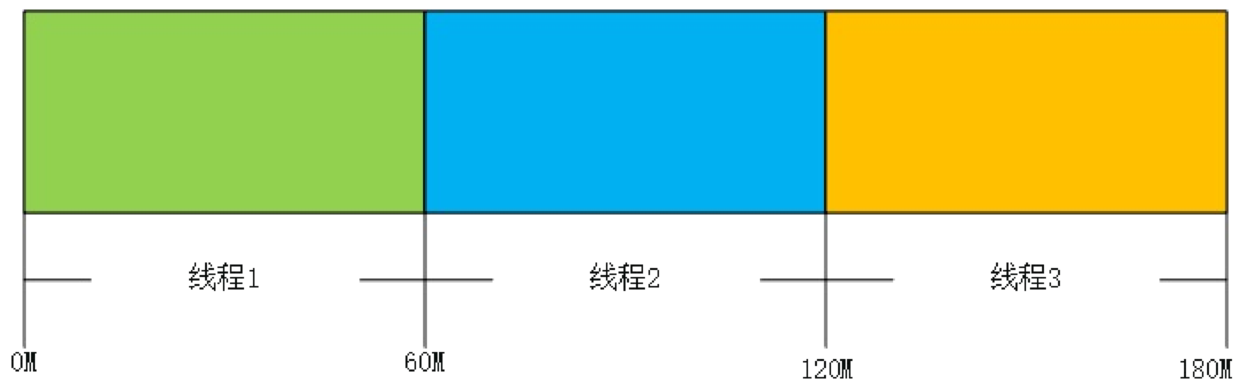
二、多线程断点续传

多线程断点续传便是在单线程的断点续传上延伸的。多线程断点续传是把整个文件分割成几个部分，每个部分由一条线程执行下载，而每一条下载线程都要实现断点续传功能。 为了实现文件分割功能，我们需要使用到httpURLconnection的另外一个方法：

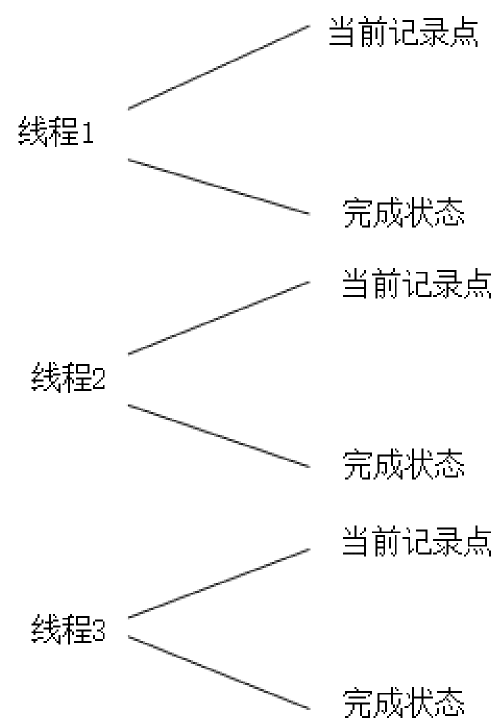
```
public int getContentLength()
```

当请求成功时，可以通过该方法获取到文件的总长度。 `每一条线程下载大小 = fileLength / THREAD_NUM`

如下图所示，描述的便是多线程的下载模型：



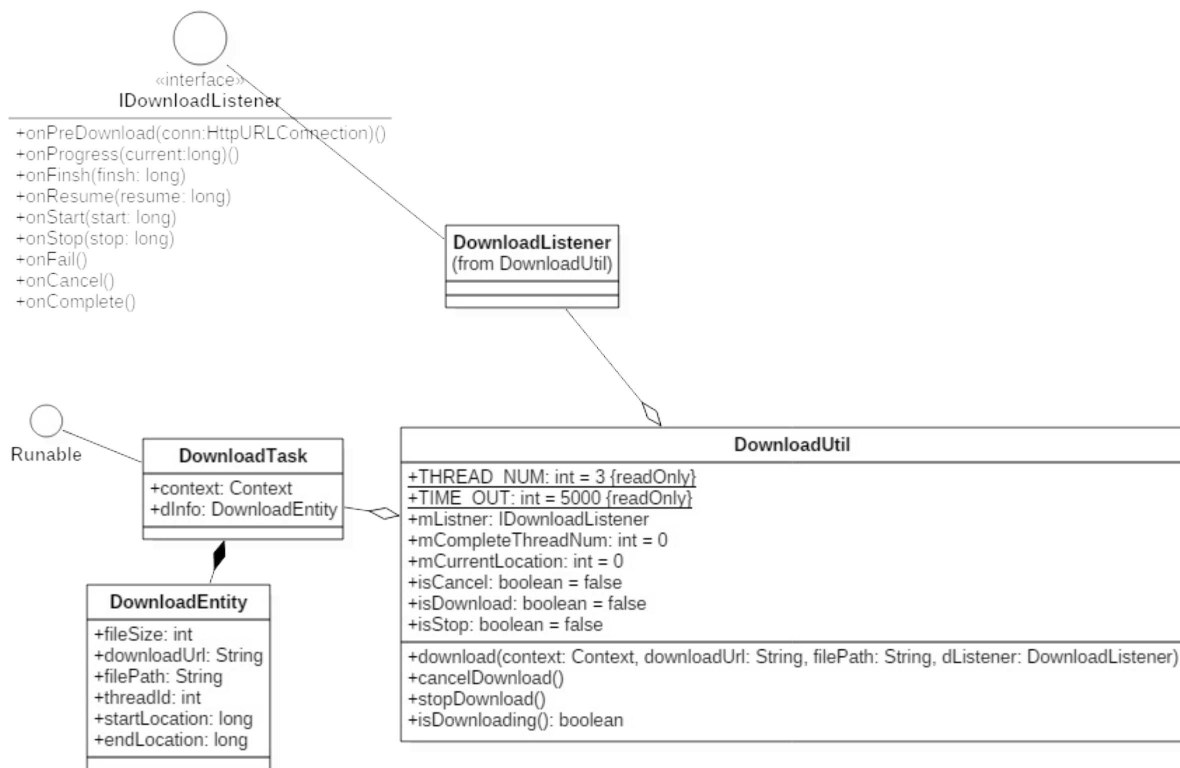
在多线程断点续传下载中，有一点需要特别注意： 由于文件是分成多个部分是被不同的线程的同时下载的，这就需要，每一条线程都分别需要有一个断点记录，和一个线程完成状态的记录；



只有所有线程的下载状态都处于完成状态时，才能表示文件已经下载完成。 实现记录的方法多种多样，我这里采用的是JDK自带的Properties类来记录下载参数。

三、断点续传结构

通过原理的了解，便可以很快的设计出断点续传工具类的基本结构图



IDownloadListener.java

```
package com.arialyy.frame.http.inf;
import java.net.HttpURLConnection;

/**
 * 在这里面编写你的业务逻辑
 */
public interface IDownloadListener {
    /**
     * 取消下载
     */
    public void onCancel();

    /**
     * 下载失败
     */
    public void onFail();

    /**
     * 下载预处理,可通过HttpURLConnection获取文件长度
```

```
         */
        public void onPreDownload(HttpURLConnection connection);

        /**
         * 下载监听
         */
        public void onProgress(long currentLocation);

        /**
         * 单一线程的结束位置
         */
        public void onChildComplete(long finishLocation);

        /**
         * 开始
         */
        public void onStart(long startLocation);

        /**
         * 子程恢复下载的位置
         */
        public void onChildResume(long resumeLocation);

        /**
         * 恢复位置
         */
        public void onResume(long resumeLocation);

        /**
         * 停止
         */
        public void onStop(long stopLocation);

        /**
         * 下载完成
         */
        public void onComplete();
    }
```

该类是下载监听接口

DownloadListener.java

```
import java.net.HttpURLConnection;

/**
 * 下载监听
 */
public class DownloadListener implements IDownloadListener {

    @Override
    public void onResume(long resumeLocation) {

    }

    @Override
    public void onCancel() {

    }

    @Override
    public void onFail() {

    }

    @Override
    public void onPreDownload(HttpURLConnection connection) {

    }

    @Override
    public void onProgress(long currentLocation) {

    }

    @Override
    public void onChildComplete(long finishLocation) {

    }
```

```
    @Override
    public void onStart(long startLocation) {

    }

    @Override
    public void onChildResume(long resumeLocation) {

    }

    @Override
    public void onStop(long stopLocation) {

    }

    @Override
    public void onComplete() {

    }
}
```

下载参数实体

```
    /**
     * 子线程下载信息类
     */
    private class DownloadEntity {
        //文件总长度
        long fileSize;
        //下载链接
        String downloadUrl;
        //线程Id
        int threadId;
        //起始下载位置
        long startLocation;
        //结束下载的文章
        long endLocation;
        //下载文件
        File tempFile;
        Context context;

        public DownloadEntity(Context context, long fileSize, String downloadUrl, File file, int threadId, long startLocation, long endLocation) {
            this.fileSize = fileSize;
            this.downloadUrl = downloadUrl;
            this.tempFile = file;
            this.threadId = threadId;
            this.startLocation = startLocation;
            this.endLocation = endLocation;
            this.context = context;
        }
    }
```

该类是下载信息配置类，每一条子线程的下载都需要一个下载实体来配置下载信息。

下载任务线程

```
    /**
     * 多线程下载任务类
```

```
     */
    private class DownLoadTask implements Runnable {
        private static final String TAG = "DownLoadTask";
        private DownloadEntity dEntity;
        private String configFPath;

        public DownLoadTask(DownloadEntity downloadInfo) {
            this.dEntity = downloadInfo;
            configFPath = dEntity.context.getFilesDir().getPath(
) + "/temp/" + dEntity.tempFile.getName() + ".properties";
        }

        @Override
        public void run() {
            try {
                L.d(TAG, "线程_" + dEntity.threadId + "_正在下载【"
 + "开始位置 : " + dEntity.startLocation + "，结束位置：" + dEntity
.endLocation + "】");
                URL url = new URL(dEntity.downloadUrl);
                HttpURLConnection conn = (HttpURLConnection) url
.openConnection();
                //在头里面请求下载开始位置和结束位置
                conn.setRequestProperty("Range", "bytes=" + dEnt
ity.startLocation + "-" + dEntity.endLocation);
                conn.setRequestMethod("GET");
                conn.setRequestProperty("Charset", "UTF-8");
                conn.setConnectTimeout(TIME_OUT);
                conn.setRequestProperty("User-Agent", "Mozilla/4
.0 (compatible; MSIE 8.0; Windows NT 5.2; Trident/4.0; .NET CLR
1.1.4322; .NET CLR 2.0.50727; .NET CLR 3.0.04506.30; .NET CLR 3.
0.4506.2152; .NET CLR 3.5.30729)");
                conn.setRequestProperty("Accept", "image/gif, im
age/jpeg, image/pjpeg, image/pjpeg, application/x-shockwave-flas
h, application/xaml+xml, application/vnd.ms-xpsdocument, applica
tion/x-ms-xbap, application/x-ms-application, application/vnd.ms
-excel, application/vnd.ms-powerpoint, application/msword, */*")
;
                conn.setReadTimeout(2000);  //设置读取流的等待时间,
必须设置该参数
                InputStream is = conn.getInputStream();
```

```
                //创建可设置位置的文件
                RandomAccessFile file = new RandomAccessFile(dEn
tity.tempFile, "rwd");
                //设置每条线程写入文件的位置
                file.seek(dEntity.startLocation);
                byte[] buffer = new byte[1024];
                int len;
                //当前子线程的下载位置
                long currentLocation = dEntity.startLocation;
                while ((len = is.read(buffer)) != -1) {
                    if (isCancel) {
                        L.d(TAG, "++++++++++ thread_" + dEntity.
threadId + "_cancel ++++++++++");
                        break;
                    }

                    if (isStop) {
                        break;
                    }

                    //把下载数据数据写入文件
                    file.write(buffer, 0, len);
                    synchronized (DownLoadUtil.this) {
                        mCurrentLocation += len;
                        mListener.onProgress(mCurrentLocation);
                    }
                    currentLocation += len;
                }
                file.close();
                is.close();

                if (isCancel) {
                    synchronized (DownLoadUtil.this) {
                        mCancelNum++;
                        if (mCancelNum == THREAD_NUM) {
                            File configFile = new File(configFPa
th);
                            if (configFile.exists()) {
                                configFile.delete();
                            }
```

```
                            if (dEntity.tempFile.exists()) {
                                dEntity.tempFile.delete();
                            }
                            L.d(TAG, "++++++++++++++++ onCancel
+++++++++++++++++");
                            isDownloading = false;
                            mListener.onCancel();
                            System.gc();
                        }
                    }
                    return;
                }

                //停止状态不需要删除记录文件
                if (isStop) {
                    synchronized (DownLoadUtil.this) {
                        mStopNum++;
                        String location = String.valueOf(current
Location);
                        L.i(TAG, "thread_" + dEntity.threadId +
"_stop, stop location ==> " + currentLocation);
                        writeConfig(dEntity.tempFile.getName() +
"_record_" + dEntity.threadId, location);
                        if (mStopNum == THREAD_NUM) {
                            L.d(TAG, "++++++++++++++++ onStop ++
++++++++++++++");
                            isDownloading = false;
                            mListener.onStop(mCurrentLocation);
                            System.gc();
                        }
                    }
                    return;
                }

                L.i(TAG, "线程【" + dEntity.threadId + "】下载完毕"
);
                writeConfig(dEntity.tempFile.getName() + "_state
_" + dEntity.threadId, 1 + "");
                mListener.onChildComplete(dEntity.endLocation);
```

```
                    mCompleteThreadNum++;
                    if (mCompleteThreadNum == THREAD_NUM) {
                        File configFile = new File(configFPath);
                        if (configFile.exists()) {
                            configFile.delete();
                        }
                        mListener.onComplete();
                        isDownloading = false;
                        System.gc();
                    }
                } catch (MalformedURLException e) {
                    e.printStackTrace();
                    isDownloading = false;
                    mListener.onFail();
                } catch (IOException e) {
                    FL.e(this, "下载失败【" + dEntity.downloadUrl + "
】" + FL.getPrintException(e));
                    isDownloading = false;
                    mListener.onFail();
                } catch (Exception e) {
                    FL.e(this, "获取流失败" + FL.getPrintException(e))
;
                    isDownloading = false;
                    mListener.onFail();
                }
            }
```

这个是每条下载子线程的下载任务类，子线程通过下载实体对每一条线程进行下载配置，由于在多断点续传的概念里，停止表示的是暂停状态，而恢复表示的是线程从记录的断点重新进行下载，所以，线程处于停止状态时是不能删除记录文件的。

下载入口

```
    /**
     * 多线程断点续传下载文件，暂停和继续
     *
     * @param context          必须添加该参数，不能使用全局变量的conte
xt
```

```
     * @param downloadUrl      下载路径
     * @param filePath         保存路径
     * @param downloadListener 下载进度监听 {@link DownloadListene
r}
     */
    public void download(final Context context, @NonNull final S
tring downloadUrl, @NonNull final String filePath,
                             @NonNull final DownloadListener downloa
dListener) {
        isDownloading = true;
        mCurrentLocation = 0;
        isStop = false;
        isCancel = false;
        mCancelNum = 0;
        mStopNum = 0;
        final File dFile = new File(filePath);
        //读取已完成的线程数
        final File configFile = new File(context.getFilesDir().g
etPath() + "/temp/" + dFile.getName() + ".properties");
        try {
            if (!configFile.exists()) { //记录文件被删除，则重新下载
                newTask = true;
                FileUtil.createFile(configFile.getPath());
            } else {
                newTask = false;
            }
        } catch (Exception e) {
            e.printStackTrace();
            mListener.onFail();
            return;
        }
        newTask = !dFile.exists();
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    mListener = downloadListener;
                    URL url = new URL(downloadUrl);
                    HttpURLConnection conn = (HttpURLConnection)
 url.openConnection();
```

```
                    conn.setRequestMethod("GET");
                    conn.setRequestProperty("Charset", "UTF-8");
                    conn.setConnectTimeout(TIME_OUT);
                    conn.setRequestProperty("User-Agent", "Mozil
la/4.0 (compatible; MSIE 8.0; Windows NT 5.2; Trident/4.0; .NET
CLR 1.1.4322; .NET CLR 2.0.50727; .NET CLR 3.0.04506.30; .NET CL
R 3.0.4506.2152; .NET CLR 3.5.30729)");
                    conn.setRequestProperty("Accept", "image/gif
, image/jpeg, image/pjpeg, image/pjpeg, application/x-shockwave-
flash, application/xaml+xml, application/vnd.ms-xpsdocument, app
lication/x-ms-xbap, application/x-ms-application, application/vn
d.ms-excel, application/vnd.ms-powerpoint, application/msword, *
/*");
                    conn.connect();
                    int len = conn.getContentLength();
                    if (len < 0) {  //网络被劫持时会出现这个问题
                        mListener.onFail();
                        return;
                    }
                    int code = conn.getResponseCode();
                    if (code == 200) {
                        int fileLength = conn.getContentLength()
;
                        //必须建一个文件
                        FileUtil.createFile(filePath);
                        RandomAccessFile file = new RandomAccess
File(filePath, "rwd");
                        //设置文件长度
                        file.setLength(fileLength);
                        mListener.onPreDownload(conn);
                        //分配每条线程的下载区间
                        Properties pro = null;
                        pro = Util.loadConfig(configFile);
                        int blockSize = fileLength / THREAD_NUM;
                        SparseArray<Thread> tasks = new SparseAr
ray<>();
                        for (int i = 0; i < THREAD_NUM; i++) {
                            long startL = i * blockSize, endL =
(i + 1) * blockSize;
                            Object state = pro.getProperty(dFile
```

```
.getName() + "_state_" + i);
                            if (state != null && Integer.parseIn
t(state + "") == 1) {  //该线程已经完成
                                mCurrentLocation += endL - start
L;
                                L.d(TAG, "++++++++++ 线程_" + i +
"_已经下载完成 ++++++++++");
                                mCompleteThreadNum++;
                                if (mCompleteThreadNum == THREAD
_NUM) {
                                    if (configFile.exists()) {
                                        configFile.delete();
                                    }
                                    mListener.onComplete();
                                    isDownloading = false;
                                    System.gc();
                                    return;
                                }
                                continue;
                            }
                            //分配下载位置
                            Object record = pro.getProperty(dFil
e.getName() + "_record_" + i);
                            if (!newTask && record != null && Lo
ng.parseLong(record + "") > 0) {        //如果有记录，则恢复下载
                                Long r = Long.parseLong(record +
"");
                                mCurrentLocation += r - startL;
                                L.d(TAG, "++++++++++ 线程_" + i +
"_恢复下载 ++++++++++");
                                mListener.onChildResume(r);
                                startL = r;
                            }
                            if (i == (THREAD_NUM - 1)) {
                                endL = fileLength;//如果整个文件的
大小不为线程个数的整数倍，则最后一个线程的结束位置即为文件的总长度
                            }
                            DownloadEntity entity = new Download
Entity(context, fileLength, downloadUrl, dFile, i, startL, endL)
;
```

```
                            DownLoadTask task = new DownLoadTask
(entity);
                            tasks.put(i, new Thread(task));
                        }
                        if (mCurrentLocation > 0) {
                            mListener.onResume(mCurrentLocation)
;
                        } else {
                            mListener.onStart(mCurrentLocation);
                        }
                        for (int i = 0, count = tasks.size(); i
< count; i++) {
                            Thread task = tasks.get(i);
                            if (task != null) {
                                task.start();
                            }
                        }
                    } else {
                        FL.e(TAG, "下载失败，返回码：" + code);
                        isDownloading = false;
                        System.gc();
                        mListener.onFail();
                    }
                } catch (IOException e) {
                    FL.e(this, "下载失败【downloadUrl:" + download
Url + "】\n【filePath:" + filePath + "】" + FL.getPrintException(
e));
                    isDownloading = false;
                    mListener.onFail();
                }
            }
        }).start();
    }
```

其实也没啥好说的，注释已经很完整了，需要注意两点 1、恢复下载时： `已下载的文件大小 = 该线程的上一次断点的位置 - 该线程起始下载位置` ；2、为了保证下载文件的完整性，只要记录文件不存在就需要重新进行下载；

四、最终效果



Demo点我

Android全局异常处理

在做android项目开发时，大家都知道如果程序出错了，会弹出来一个强制退出的弹出框，这个本身没什么问题，但是这个UI实在是太丑了，别说用户接受不了，就连我们自己本身可能都接受不了。虽然我们在发布程序时总会经过仔细的测试，但是难免会碰到预料不到的错误。

今天就来自定义一个程序出错时的处理，类似iphone的闪退。(虽然闪退也是用户不愿意看到的，但是在用户体验上明显比那个原生的弹窗好多了)

废话不多说，直接上代码：

CrashHandler

```
/**
 * 自定义的 异常处理类 , 实现了 UncaughtExceptionHandler接口
 *
 */
public class CrashHandler implements UncaughtExceptionHandler {

    // 需求是 整个应用程序 只有一个 MyCrash-Handler
    private static CrashHandler INSTANCE ;
    private Context context;

    //1.私有化构造方法
    private CrashHandler(){

    }

    public static synchronized CrashHandler getInstance(){
        if (INSTANCE == null)
            INSTANCE = new CrashHandler();
        return INSTANCE;
    }

    public void init(Context context){
        this.context = context;
    }


    public void uncaughtException(Thread arg0, Throwable arg1) {

        System.out.println("程序挂掉了 ");
        // 在此可以把用户手机的一些信息以及异常信息捕获并上传,由于UMeng在
这方面有很程序的api接口来调用，故没有考虑

        //干掉当前的程序
        android.os.Process.killProcess(android.os.Process.myPid(
));
    }

}
```

CrashApplication

```
/**
 * 在开发应用时都会和Activity打交道，而Application使用的就相对较少了。
 * Application是用来管理应用程序的全局状态的，比如载入资源文件。
 * 在应用程序启动的时候Application会首先创建，然后才会根据情况(Intent)启动相应的Activity或者Service。
 * 在本文将在Application中注册未捕获异常处理器。
 */
public class CrashApplication extends Application {
    @Override
    public void onCreate() {
        super.onCreate();
        CrashHandler handler = CrashHandler.getInstance();
        handler.init(getApplicationContext());
        Thread.setDefaultUncaughtExceptionHandler(handler);
    }
}
```

在AndroidManifest.xml中注册

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.wp.activity" android:versionCode="1" android:versionName="1.0">
    <application android:icon="@drawable/icon" android:label="@string/app_name"
        android:name=".CrashApplication" android:debuggable="true">
        <activity android:name=".MainActivity" android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
    <uses-sdk android:minSdkVersion="8" />
</manifest>
```

至此，可以测试下在出错的时候程序会直接闪退，并杀死后台进程。当然也可以自定义一些比较友好的出错UI提示，进一步提升用户体验。

一、MVP概述

MVP，全称 **Model-View-Presenter**，即模型-视图-层现器。

提到MVP，就必须要先介绍一下它的前辈MVC，因为MVP正是基于MVC的基础发展而来的。两个之间的关系也是源远流长。

MVC，全称**Model-View-Controller**，即模型-视图-控制器。 具体如下：

View：对应于布局文件

Model：业务逻辑和实体模型

Controllor：对应于Activity

但是View对应于布局文件，其实能做的事情特别少，实际上关于该布局文件中的数据绑定的操作，事件处理的代码都在Activity中，造成了Activity既像View又像Controller，使得Activity变得臃肿。

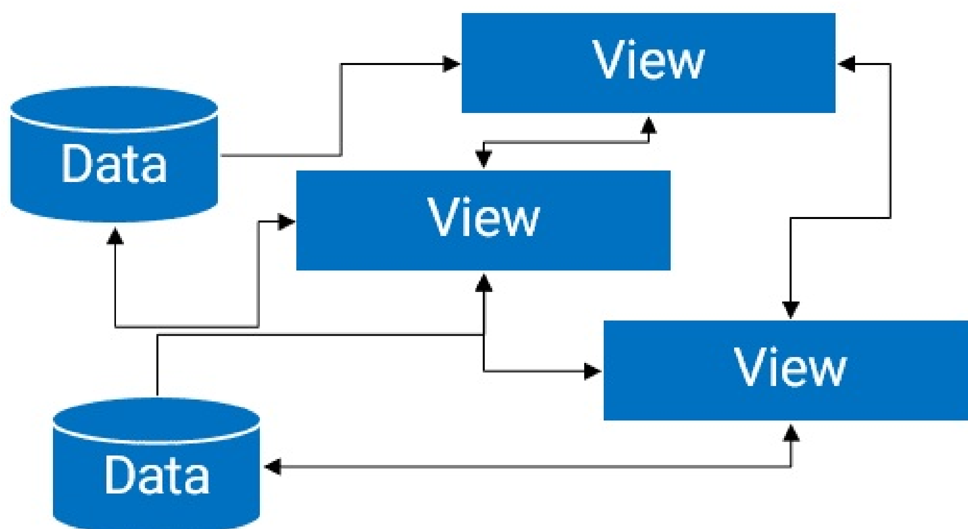
而当将架构改为MVP以后，Presenter的出现，将Actvity视为View层，Presenter负责完成View层与Model层的交互。现在是这样的：

View 对应于**Activity**，负责**View**的绘制以及与用户交互

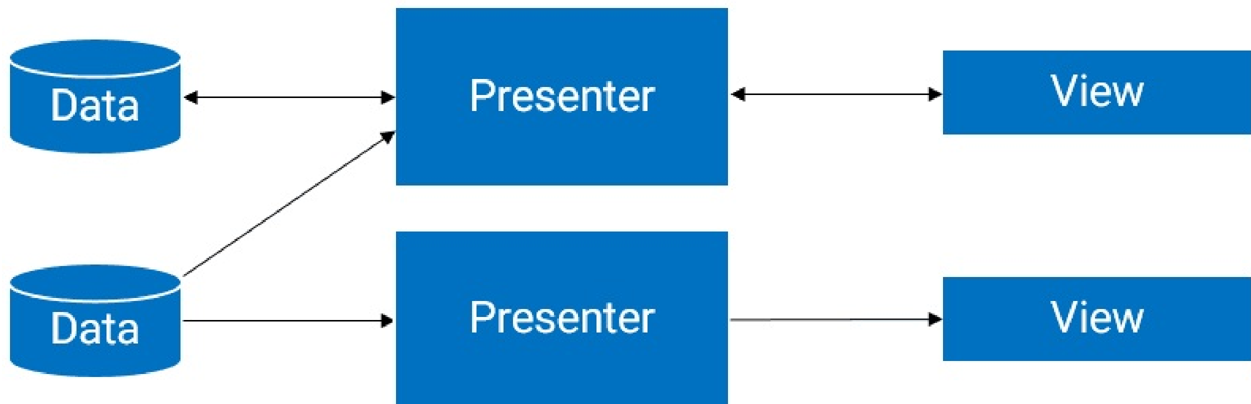
Model 依然是业务逻辑和实体模型

Presenter 负责完成View于Model间的交互

下面两幅图通过数据与视图之间的交互清楚地展示了这种变化：



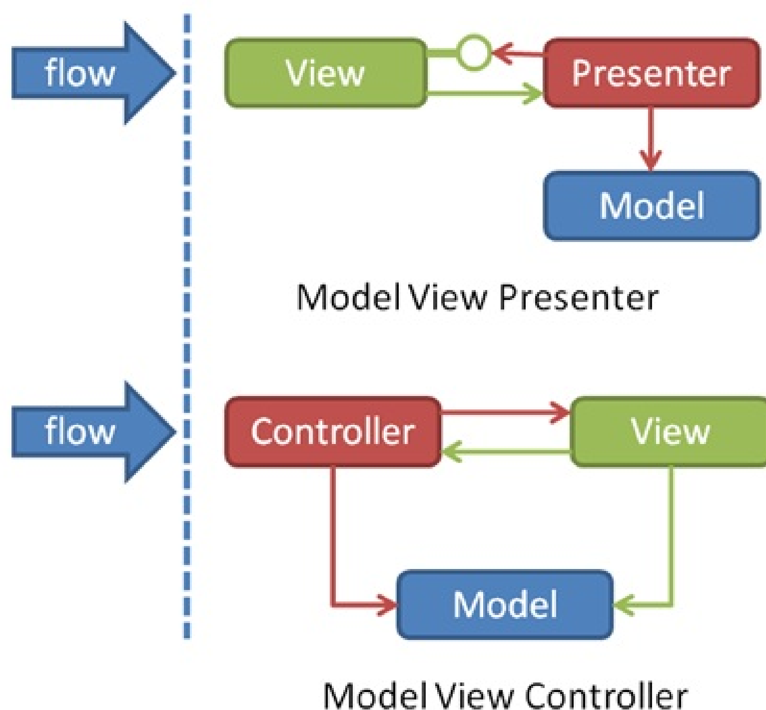
MVC模式下实际上就是Activty与Model之间交互，View完全独立出来了。



MVP模式通过Presenter实现数据和视图之间的交互，简化了Activity的职责。同时即避免了View和Model的直接联系，又通过Presenter实现两者之间的沟通。

总结：**MVP**模式减少了**Activity**的职责，简化了**Activity**中的代码，将复杂的逻辑代码提取到了**Presenter**中进行处理，模块职责划分明显，层次清晰。与之对应的好处就是，耦合度更低，更方便的进行测试。

MVC和MVP的区别

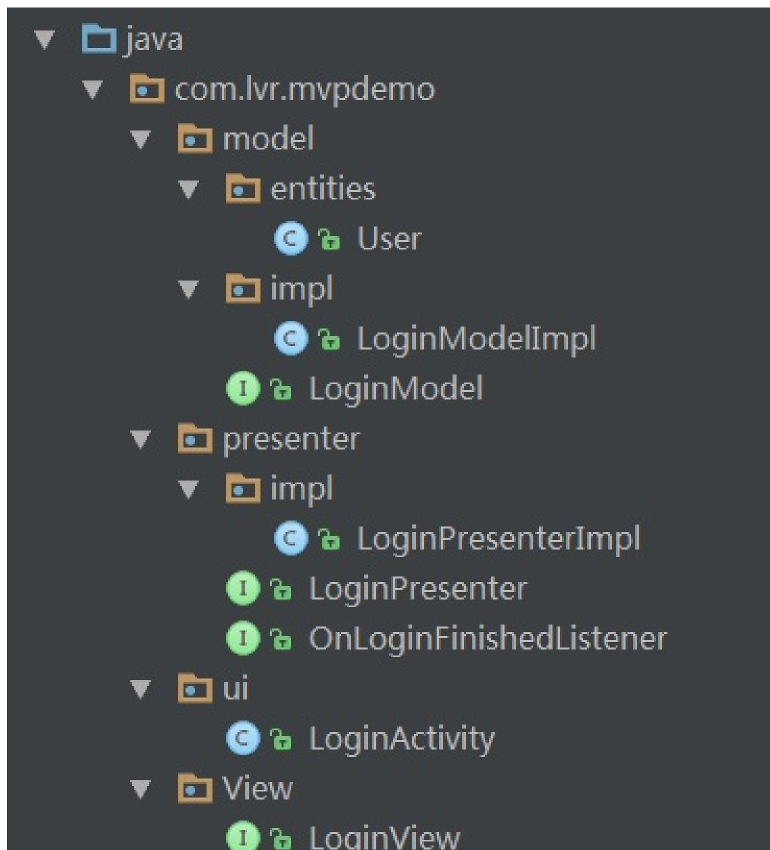


MVC中是允许**Model**和**View**进行交互的，而**MVP**中很明显，**Model**与**View**之间的交互由**Presenter**完成。还有一点就是**Presenter**与**View**之间的交互是通过接口的。

还有一点注意：**MVC**中**V**对应的是布局文件，**MVP**中**V**对应的是**Activity**。

二、MVP的简单使用

大多数MVP模式的示例都使用登录案例进行介绍。因为简单方便，同时能提现出MVP的特点。今天我们也以此例进行学习。 使用MVP的好处之一就是模块职责划分明显，层次清晰。 该例的结构图即可展现此优点。



1.Model层

在本例中，M0del层负责对从登录页面获取地帐号密码进行验证（一般需要请求服务器进行验证，本例直接模拟这一过程）。 从上图的包结构图中可以看出，Model层包含内容：

①实体类bean

②接口，表示Model层所要执行的业务逻辑

③接口实现类，具体实现业务逻辑，包含的一些主要方法

下面以代码的形式一一展开。

①实体类bean

```
public class User {
    private String password;
    private String username;

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    @Override
    public String toString() {
        return "User{" +
                "password='" + password + '\'' +
                ", username='" + username + '\'' +
                '}';
    }
}
```

封装了用户名、密码，方便数据传递。

②接口

```
public interface LoginModel {
    void login(User user, OnLoginFinishedListener listener);
}
```

其中OnLoginFinishedListener 是presenter层的接口，方便实现回调presenter，通知presenter业务逻辑的返回结果，具体在presenter层介绍。

③接口实现类

```
public class LoginModelImpl implements LoginModel {
    @Override
    public void login(User user, final OnLoginFinishedListener l
istener) {
        final String username = user.getUsername();
        final String password = user.getPassword();
        new Handler().postDelayed(new Runnable() {
            @Override public void run() {
                boolean error = false;
                if (TextUtils.isEmpty(username)){
                    listener.onUsernameError();//model层里面回调li
stener
                    error = true;
                }
                if (TextUtils.isEmpty(password)){
                    listener.onPasswordError();
                    error = true;
                }
                if (!error){
                    listener.onSuccess();
                }
            }
        }, 2000);
    }
}
```

实现Model层逻辑：延时模拟登陆（2s），如果用户名或者密码为空则登陆失败，否则登陆成功。

2.View层

视图：将Modle层请求的数据呈现给用户。一般的视图都只是包含用户界面(UI)，而不包含界面逻辑，界面逻辑由Presenter来实现。

从上图的包结构图中可以看出，View包含内容：

①接口，上面我们说过Presenter与View交互是通过接口。其中接口中方法的定义是根据Activity用户交互需要展示的控件确定的。

②接口实现类，将上述定义的接口中的方法在Activity中对应实现具体操作。

下面以代码的形式一一展开。

①接口

```
public interface LoginView {
    //login是个耗时操作，我们需要给用户一个友好的提示，一般就是操作Progre
ssBar
    void showProgress();

    void hideProgress();
   //login当然存在登录成功与失败的处理，失败给出提示
    void setUsernameError();

    void setPasswordError();
   //login成功，也给个提示
    void showSuccess();
}
```

上述5个方法都是presenter根据model层返回结果需要view执行的对应的操作。

②接口实现类

即对应的登录的Activity，需要实现LoginView接口。

```
public class LoginActivity extends AppCompatActivity implements
LoginView, View.OnClickListener {
    private ProgressBar progressBar;
    private EditText username;
    private EditText password;
    private LoginPresenter presenter;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_login);
```

```
        progressBar = (ProgressBar) findViewById(R.id.progress);
        username = (EditText) findViewById(R.id.username);
        password = (EditText) findViewById(R.id.password);
        findViewById(R.id.button).setOnClickListener(this);
       //创建一个presenter对象，当点击登录按钮时，让presenter去调用mode
l层的login()方法，验证帐号密码
        presenter = new LoginPresenterImpl(this);
    }

    @Override
    protected void onDestroy() {
        presenter.onDestroy();
        super.onDestroy();
    }

    @Override
    public void showProgress() {
        progressBar.setVisibility(View.VISIBLE);
    }

    @Override
    public void hideProgress() {
        progressBar.setVisibility(View.GONE);
    }

    @Override
    public void setUsernameError() {
        username.setError(getString(R.string.username_error));
    }

    @Override
    public void setPasswordError() {
        password.setError(getString(R.string.password_error));
    }

    @Override
    public void showSuccess() {
         progressBar.setVisibility(View.GONE);
        Toast.makeText(this,"login success",Toast.LENGTH_SHORT).
show();
```

```
    }

    @Override
    public void onClick(View v) {
        User user = new User();
        user.setPassword(password.getText().toString());
        user.setUsername(username.getText().toString());
        presenter.validateCredentials(user);
    }

}
```

View层实现Presenter层需要调用的控件操作，方便Presenter层根据Model层返回的结果进行操作View层进行对应的显示。

3.Presenter层

Presenter是用作Model和View之间交互的桥梁。 从上图的包结构图中可以看出，Presenter包含内容：

①接口，包含Presenter需要进行Model和View之间交互逻辑的接口，以及上面提到的Model层数据请求完成后回调的接口。

②接口实现类，即实现具体的Presenter类逻辑。

下面以代码的形式一一展开。

①接口

```
public interface OnLoginFinishedListener {
    void onUsernameError();

    void onPasswordError();

    void onSuccess();
}
```

当Model层得到请求的结果，需要回调Presenter层，让Presenter层调用View层的接口方法。

```
public interface LoginPresenter {
    void validateCredentials(User user);

    void onDestroy();
}
```

登陆的Presenter 的接口，实现类为LoginPresenterImpl，完成登陆的验证，以及销毁当前view。

②接口实现类

```
public class LoginPresenterImpl implements LoginPresenter, OnLog
inFinishedListener {
    private LoginView loginView;
    private LoginModel loginModel;

    public LoginPresenterImpl(LoginView loginView) {
        this.loginView = loginView;
        this.loginModel = new LoginModelImpl();
    }

    @Override
    public void validateCredentials(User user) {
        if (loginView != null) {
            loginView.showProgress();
        }

        loginModel.login(user, this);
    }

    @Override
    public void onDestroy() {
        loginView = null;
    }

    @Override
    public void onUsernameError() {
        if (loginView != null) {
            loginView.setUsernameError();
```

```
                loginView.hideProgress();
            }
        }

        @Override
        public void onPasswordError() {
            if (loginView != null) {
                loginView.setPasswordError();
                loginView.hideProgress();
            }
        }

        @Override
        public void onSuccess() {
            if (loginView != null) {
                loginView.showSuccess();
            }
        }
    }
```

由于presenter完成二者的交互，那么肯定需要二者的实现类（通过传入参数，或者new）。

presenter里面有个OnLoginFinishedListener， 其在Presenter层实现，给Model层回调，更改View层的状态， 确保 Model层不直接操作View层。

示例展示：

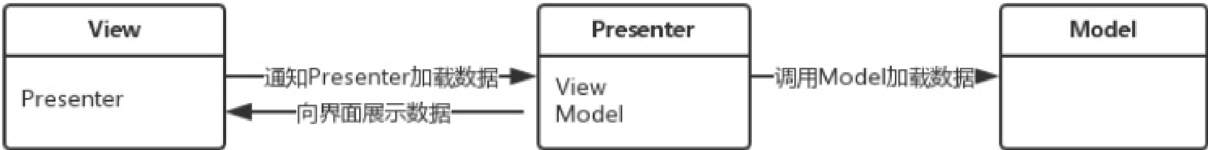

4G 0K/s
开发模式
HD 84%
MVPdemo
帐号
密码
登录

代码地址

三、总结

MVP模式的整个核心流程：

View与Model并不直接交互，而是使用Presenter作为View与Model之间的桥梁。其中Presenter中同时持有View层的Interface的引用以及Model层的引用，而View层持有Presenter层引用。当View层某个界面需要展示某些数据的时候，首先会调用Presenter层的引用，然后Presenter层会调用Model层请求数据，当Model层数据加载成功之后会调用Presenter层的回调方法通知Presenter层数据加载情况，最后Presenter层再调用View层的接口将加载后的数据展示给用户。



Binder原理

1、概述

Android系统中，涉及到多进程间的通信底层都是依赖于Binder IPC机制。例如当进程A中的Activity要向进程B中的Service通信，这便需要依赖于Binder IPC。不仅于此，整个Android系统架构中，大量采用了Binder机制作为IPC（进程间通信，Interprocess Communication）方案。

当然也存在部分其他的IPC方式，如管道、SystemV、Socket等。那么Android为什么不使用这些原有的技术，而是要使开发一种新的叫Binder的进程间通信机制呢？

为什么要使用Binder？

性能方面

在移动设备上（性能受限制的设备，比如要省电），广泛地使用跨进程通信对通信机制的性能有严格的要求，Binder相对于传统的Socket方式，更加高效。**Binder**数据拷贝只需要一次，而管道、消息队列、**Socket**都需要**2**次，共享内存方式一次内存拷贝都不需要，但实现方式又比较复杂。

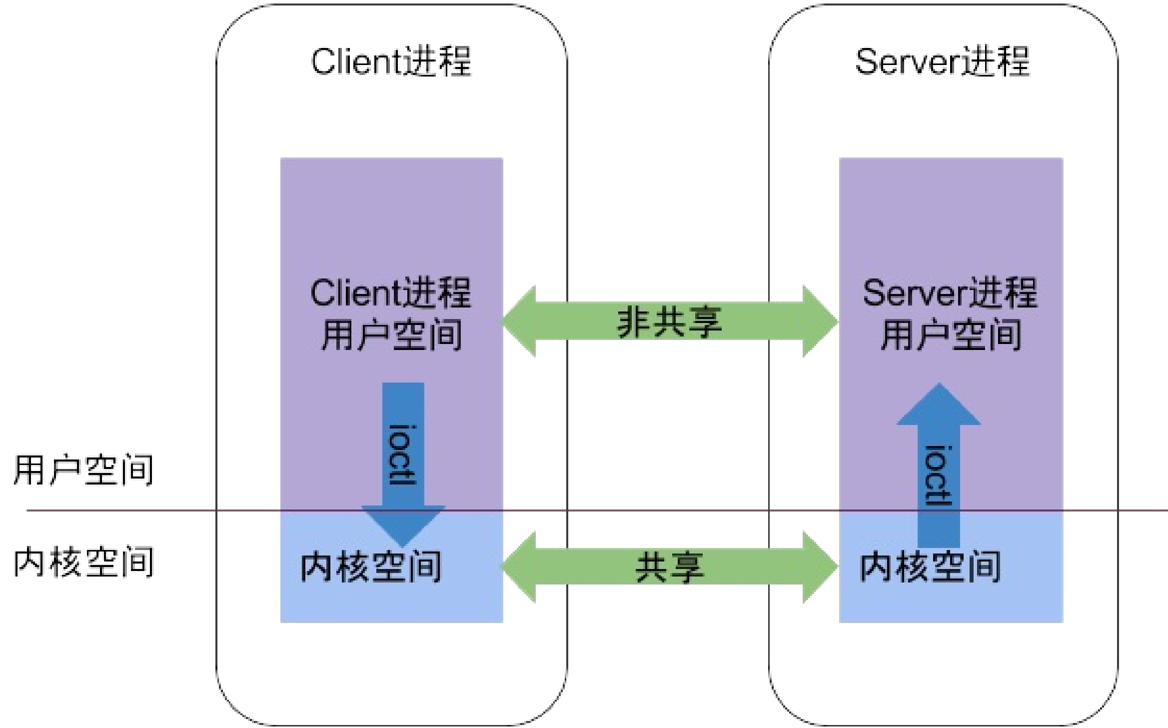
安全方面

传统的进程通信方式对于通信双方的身份并没有做出严格的验证，比如Socket通信的IP地址是客户端手动填入，很容易进行伪造。然而，Binder机制从协议本身就支持对通信双方做身份校检，从而大大提升了安全性。

2、 Binder

IPC原理

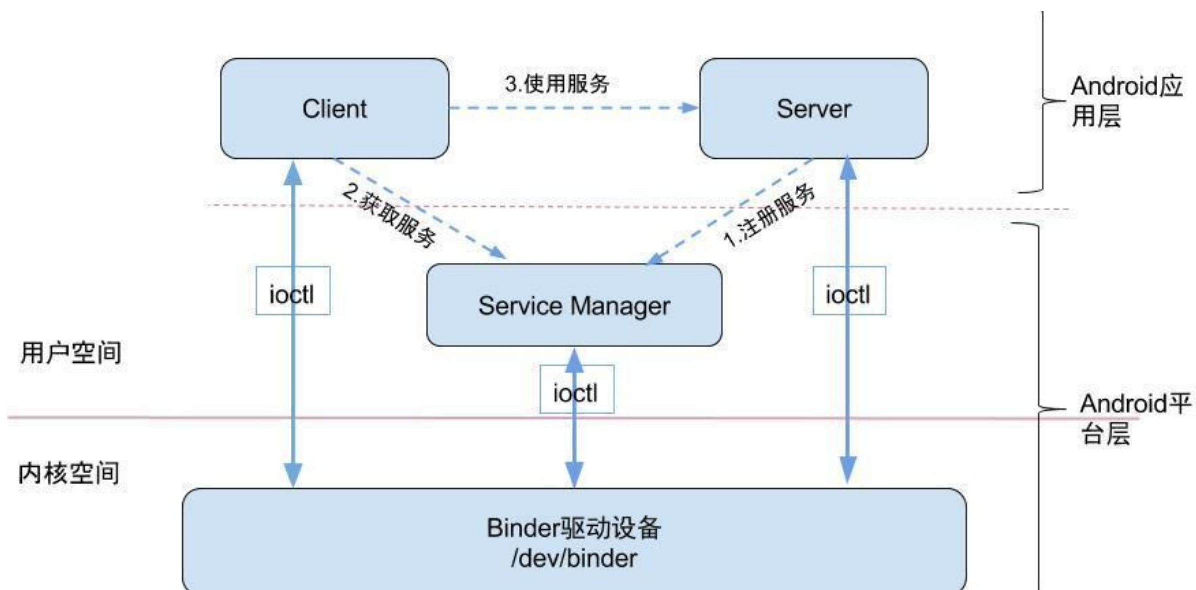
从进程角度来看IPC（Interprocess Communication）机制



每个Android的进程，只能运行在自己进程所拥有的虚拟地址空间。例如，对应一个4GB的虚拟地址空间，其中3GB是用户空间，1GB是内核空间。当然内核空间的大小是可以通过参数配置调整的。对于用户空间，不同进程之间是不能共享的，而内核空间却是可共享的。Client进程向Server进程通信，恰恰是利用进程间可共享的内核内存空间来完成底层通信工作的。Client端与Server端进程往往采用ioctl等方法与内核空间的驱动进行交互。

Binder原理

Binder通信采用C/S架构，从组件视角来说，包含Client、Server、ServiceManager以及Binder驱动，其中ServiceManager用于管理系统中的各种服务。架构图如下所示：



Binder通信的四个角色

Client进程：使用服务的进程。

Server进程：提供服务的进程。

ServiceManager进程：ServiceManager的作用是将字符形式的Binder名字转化成Client中对该Binder的引用，使得Client能够通过Binder名字获得对Server中Binder实体的引用。

Binder驱动：驱动负责进程之间Binder通信的建立，Binder在进程之间的传递，Binder引用计数管理，数据包在进程之间的传递和交互等一系列底层支持。

Binder运行机制

图中Client/Server/ServiceManage之间的相互通信都是基于Binder机制。既然基于Binder机制通信，那么同样也是C/S架构，则图中的3大步骤都有相应的Client端与Server端。

注册服务**(addService)**：Server进程要先注册Service到ServiceManager。该过程：Server是客户端，ServiceManager是服务端。

获取服务**(getService)**：Client进程使用某个Service前，须先向ServiceManager中获取相应的Service。该过程：Client是客户端，ServiceManager是服务端。

使用服务：Client根据得到的Service信息建立与Service所在的Server进程通信的通路，然后就可以直接与Service交互。该过程：Client是客户端，Server是服务端。

图中的Client，Server，Service Manager之间交互都是虚线表示，是由于它们彼此之间不是直接交互的，而是都通过与Binder驱动进行交互的，从而实现IPC通信（Interprocess Communication）方式。其中Binder驱动位于内核空间，Client，Server，Service Manager位于用户空间。Binder驱动和Service Manager可以看做是Android平台的基础架构，而Client和Server是Android的应用层，开发人员只需自定义实现Client、Server端，借助Android的基本平台架构便可以直接进行IPC通信。

Binder运行的实例解释

首先我们看看我们的程序跨进程调用系统服务的简单示例，实现浮动窗口部分代码：

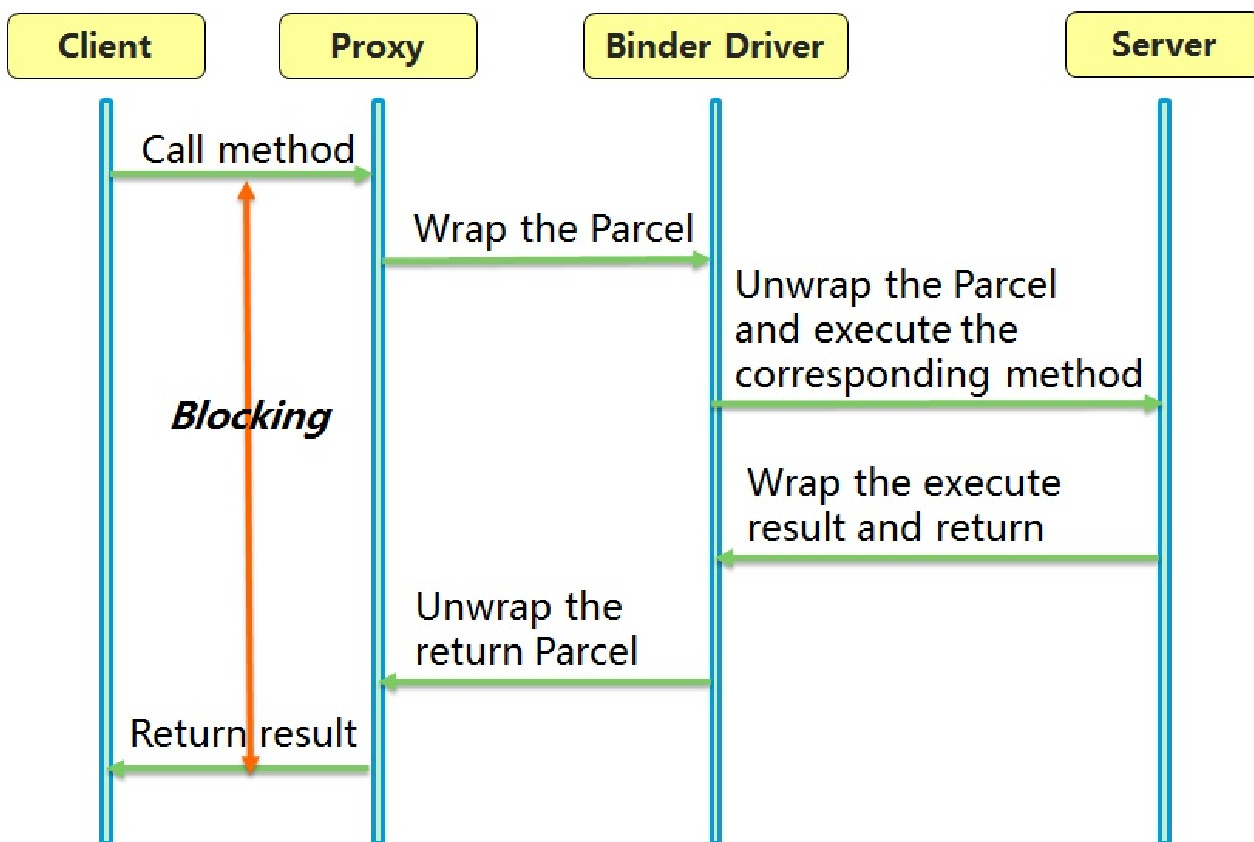
```
//获取WindowManager服务引用
WindowManager wm = (WindowManager) getSystemService(getApplicati
on().WINDOW_SERVICE);
//布局参数layoutParams相关设置略...
View view = LayoutInflater.from(getApplication()).inflate(R.layo
ut.float_layout, null);
//添加view
wm.addView(view, layoutParams);
```

注册服务**(addService)**： 在Android开机启动过程中，Android会初始化系统的各种Service，并将这些Service向ServiceManager注册（即让ServiceManager管理）。这一步是系统自动完成的。

获取服务**(getService)**： 客户端想要得到具体的Service直接向ServiceManager要即可。客户端首先向ServiceManager查询得到具体的Service引用，通常是Service引用的代理对象，对数据进行一些处理操作。即第2行代码中，得到的wm是WindowManager对象的引用。

使用服务： 通过这个引用向具体的服务端发送请求，服务端执行完成后就返回。即第6行调用WindowManager的addView函数，将触发远程调用，调用的是运行在systemServer进程中的WindowManager的addView函数。

使用服务的具体执行过程



1. Client通过获得一个Server的代理接口，对Server进行调用。
2. 代理接口中定义的方法与Server中定义的方法是一一对应的。
3. Client调用某个代理接口中的方法时，代理接口的方法会将Client传递的参数打包成Parcel对象。
4. 代理接口将Parcel发送给内核中的Binder Driver。
5. Server会读取Binder Driver中的请求数据，如果是发送给自己的，解包Parcel对象，处理并将结果返回。
6. 整个的调用过程是一个同步过程，在Server处理的时候，Client会Block住。因此**Client**调用过程不应在主线程。

AIDL的使用

1.AIDL的简介

AIDL (Android Interface Definition Language) 是一种接口定义语言，用于生成可以在Android设备上两个进程之间进行进程间通信(Interprocess Communication, IPC)的代码。如果在一个进程中（例如Activity）要调用另一个进程中（例如Service）对象的操作，就可以使用AIDL生成可序列化的参数，来完成进程间通信。

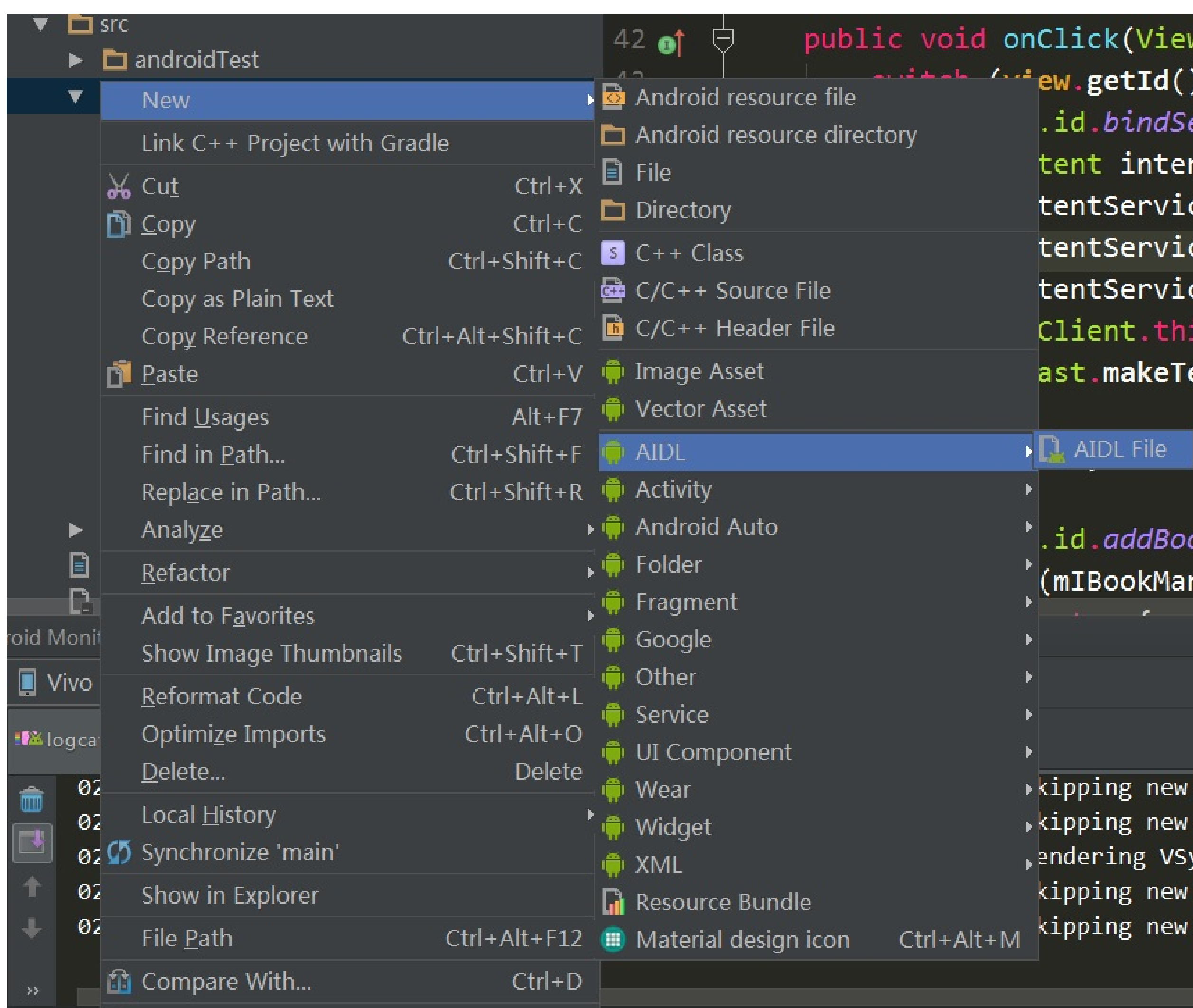
简言之，AIDL能够实现进程间通信，其内部是通过Binder机制来实现的，后面会具体介绍，现在先介绍AIDL的使用。

2.AIDL的具体使用

AIDL的实现一共分为三部分，一部分是客户端，调用远程服务。一部分是服务端，提供服务。最后一部分，也是最关键的是AIDL接口，用来传递的参数，提供进程间通信。

先在服务端创建AIDL部分代码。

AIDL文件 通过如下方式新建一个AIDL文件



默认生成格式

```
interface IBookManager {
    /**
     * Demonstrates some basic types that you can use as paramet
ers
     * and return values in AIDL.
     */
    void basicTypes(int anInt, long aLong, boolean aBoolean, flo
at aFloat,
            double aDouble, String aString);
}
```

默认如下格式，由于本例要操作Book类，实现两个方法，添加书本和返回书本列表。

定义一个**Book**类，实现**Parcelable**接口。

```
public class Book implements Parcelable {
    public int bookId;
    public String bookName;

    public Book() {
    }

    public Book(int bookId, String bookName) {
        this.bookId = bookId;
        this.bookName = bookName;
    }

    public int getBookId() {
        return bookId;
    }

    public void setBookId(int bookId) {
        this.bookId = bookId;
    }

    public String getBookName() {
        return bookName;
    }
```

```
    public void setBookName(String bookName) {
        this.bookName = bookName;
    }

    @Override
    public int describeContents() {
        return 0;
    }

    @Override
    public void writeToParcel(Parcel dest, int flags) {
        dest.writeInt(this.bookId);
        dest.writeString(this.bookName);
    }

    protected Book(Parcel in) {
        this.bookId = in.readInt();
        this.bookName = in.readString();
    }

    public static final Parcelable.Creator<Book> CREATOR = new P
arcelable.Creator<Book>() {
        @Override
        public Book createFromParcel(Parcel source) {
            return new Book(source);
        }

        @Override
        public Book[] newArray(int size) {
            return new Book[size];
        }
    };
}
```

由于AIDL只支持数据类型:基本类型（int，long，char，boolean等），String，CharSequence，List，Map，其他类型必须使用import导入，即使它们可能在同一个包里，比如上面的Book。

最终**IBookManager.aidl** 的实现

```
// Declare any non-default types here with import statements
import com.lvr.aidldemo.Book;

interface IBookManager {
    /**
     * Demonstrates some basic types that you can use as paramet
ers
     * and return values in AIDL.
     */
    void basicTypes(int anInt, long aLong, boolean aBoolean, flo
at aFloat,
            double aDouble, String aString);

    void addBook(in Book book);

    List<Book> getBookList();

}
```

注意：如果自定义的**Parcelable**对象，必须创建一个和它同名的**AIDL**文件，并在其中声明它为**parcelable**类型。

Book.aidl

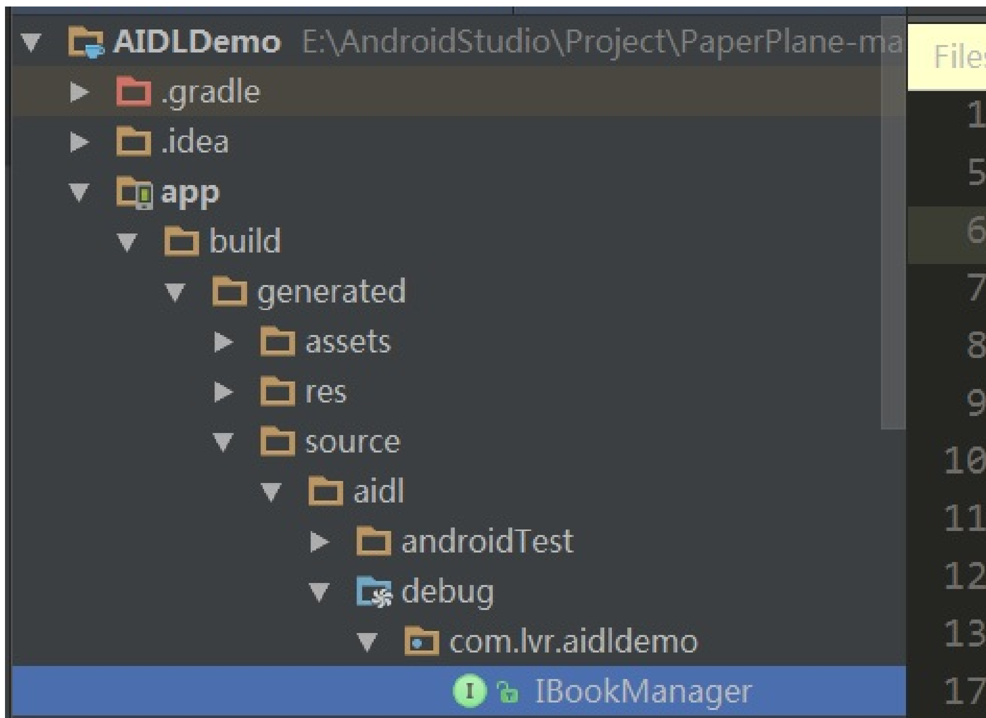
```
// Book.aidl
package com.lvr.aidldemo;

parcelable Book;
```

以上就是AIDL部分的实现，一共三个文件。

然后Make Project ，SDK为自动为我们生成对应的Binder类。

在如下路径下：



其中该接口中有个重要的内部类Stub ，继承了Binder 类，同时实现了IBookManager接口。 这个内部类是接下来的关键内容。

```
public static abstract class Stub extends android.os.Binder impl
ements com.lvr.aidldemo.IBookManager{}
```

服务端 服务端首先要创建一个Service用来监听客户端的连接请求。然后在Service中实现Stub 类，并定义接口中方法的具体实现。

```
//实现了AIDL的抽象函数
private IBookManager.Stub mbinder = new IBookManager.Stub() {
    @Override
    public void basicTypes(int anInt, long aLong, boolean aBoolean, float aFloat, double aDouble, String aString) throws RemoteException {
        //什么也不做
    }

    @Override
    public void addBook(Book book) throws RemoteException {
        //添加书本
        if (!mBookList.contains(book)) {
            mBookList.add(book);
        }
    }

    @Override
    public List<Book> getBookList() throws RemoteException {
        return mBookList;
    }
};
```

当客户端连接服务端，服务端就会调用如下方法：

```
public IBinder onBind(Intent intent) {
    return mbinder;
}
```

就会把Stub实现对象返回给客户端，该对象是个Binder对象，可以实现进程间通信。 本例就不真实模拟两个应用之间的通信，而是让Service另外开启一个进程来模拟进程间通信。

```
<service
    android:name=".MyService"
    android:process=":remote">
    <intent-filter>
        <category android:name="android.intent.category.DEFAULT"
 />
        <action android:name="com.lvr.aidldemo.MyService" />
    </intent-filter>
</service>
```

`android:process=":remote"` 设置为另一个进程。 `<action android:name="com.lvr.aidldemo.MyService"/>` 是为了能让其他apk隐式bindService。通过隐式调用的方式来连接**service**，需要把**category**设为**default**，这是因为，隐式调用的时候，**intent**中的**category**默认会被设置为**default**。

客户端

首先将服务端工程中的**aidl**文件夹下的内容整个拷贝到客户端工程的对应位置下，由于本例的使用在一个应用中，就不需要拷贝了，其他情况一定不要忘记这一步。

客户端需要做的事情比较简单，首先需要绑定服务端的Service。

```
Intent intentService = new Intent();
intentService.setAction("com.lvr.aidldemo.MyService");
intentService.setPackage(getPackageName());
intentService.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
MyClient.this.bindService(intentService, mServiceConnection, BIN
D_AUTO_CREATE);
Toast.makeText(getApplicationContext(), "绑定了服务", Toast.LENGTH
_SHORT).show();
```

将服务端返回的Binder对象转换成AIDL接口所属的类型，接着就可以调用AIDL中的方法了。

```
if (mIBookManager != null) {
    try {
        mIBookManager.addBook(new Book(18, "新添加的书"));
        Toast.makeText(getApplicationContext(), mIBookManager.ge
tBookList().size() + "", Toast.LENGTH_SHORT).show();
    } catch (RemoteException e) {
        e.printStackTrace();
    }
}
```

3.AIDL的工作原理

Binder机制的运行主要包括三个部分：注册服务、获取服务和使用服务。 其中注册服务和获取服务的流程涉及C的内容，由于个人能力有限，就不予介绍了。

本篇文章主要介绍使用服务时，AIDL的工作原理。

①.Binder对象的获取

Binder是实现跨进程通信的基础，那么Binder对象在服务端和客户端是共享的，是同一个Binder对象。在客户端通过Binder对象获取实现了IInterface接口的对象来调用远程服务，然后通过Binder来实现参数传递。

那么如何维护实现了IInterface接口的对象和获取Binder对象呢？

服务端获取**Binder**对象并保存**IInterface**接口对象 Binder中两个关键方法：

```
public class Binder implement IBinder {
    void attachInterface(IInterface plus, String descriptor)

    IInterface queryLocalInterface(Stringdescriptor) //从IBinder
中继承而来
    ..........................
}
```

Binder具有被跨进程传输的能力是因为它实现了IBinder接口。系统会为每个实现了该接口的对象提供跨进程传输，这是系统给我们的一个很大的福利。

Binder具有的完成特定任务的能力是通过它的**IInterface**的对象获得的，我们可以简单理解attachInterface方法会将（descriptor，plus）作为（key,value）对存入Binder对象中的一个Map对象中，Binder对象可通过attachInterface方法持有一个IInterface对象（即plus）的引用，并依靠它获得完成特定任务的能力。queryLocalInterface方法可以认为是根据key值（即参数 descriptor）查找相应的IInterface对象。

在服务端进程，通过实现`private IBookManager.Stub mbinder = new IBookManager.Stub() {}`抽象类，获得Binder对象。 并保存了IInterface对象。

```
public Stub() {
    this.attachInterface(this, DESCRIPTOR);
}
```

客户端获取**Binder**对象并获取**IInterface**接口对象

通过bindService获得Binder对象

```
MyClient.this.bindService(intentService, mServiceConnection, BIN
D_AUTO_CREATE);
```

然后通过Binder对象获得IInterface对象。

```
private ServiceConnection mServiceConnection = new ServiceConnec
tion() {
    @Override
    public void onServiceConnected(ComponentName name, IBinder b
inder) {
        //通过服务端onBind方法返回的binder对象得到IBookManager的实例，
得到实例就可以调用它的方法了
        mIBookManager = IBookManager.Stub.asInterface(binder);
    }

    @Override
    public void onServiceDisconnected(ComponentName name) {
        mIBookManager = null;
    }
};
```

其中 `asInterface(binder)` 方法如下：

```
public static com.lvr.aidldemo.IBookManager asInterface(android.
os.IBinder obj) {
    if ((obj == null)) {
        return null;
    }
    android.os.IInterface iin = obj.queryLocalInterface(DESCRIPT
OR);
    if (((iin != null) && (iin instanceof com.lvr.aidldemo.IBook
Manager))) {
        return ((com.lvr.aidldemo.IBookManager) iin);
    }
    return new com.lvr.aidldemo.IBookManager.Stub.Proxy(obj);
}
```

先通过 `queryLocalInterface(DESCRIPTOR);` 查找到对应的IInterface对象，然后判断对象的类型，如果是同一个进程调用则返回IBookManager对象，由于是跨进程调用则返回Proxy对象，即Binder类的代理对象。

②.调用服务端方法

获得了Binder类的代理对象，并且通过代理对象获得了IInterface对象，那么就可以调用接口的具体实现方法了，来实现调用服务端方法的目的。

以addBook方法为例，调用该方法后，客户端线程挂起，等待唤醒：

```
    @Override public void addBook(com.lvr.aidldemo.Book book) th
rows android.os.RemoteException
    {
        ..........
        //第一个参数：识别调用哪一个方法的ID
        //第二个参数：Book的序列化传入数据
        //第三个参数：调用方法后返回的数据
        //最后一个不用管
        mRemote.transact(Stub.TRANSACTION_addBook, _data, _reply
, 0);
        _reply.readException();
    }
    ..........
}
```

省略部分主要完成对添加的Book对象进行序列化工作，然后调用 `transact` 方法。

Proxy对象中的transact调用发生后，会引起系统的注意，系统意识到Proxy对象想找它的真身Binder对象（系统其实一直存着Binder和Proxy的对应关系）。于是系统将这个请求中的数据转发给Binder对象，Binder对象将会在onTransact中收到Proxy对象传来的数据，于是它从data中取出客户端进程传来的数据，又根据第一个参数确定想让它执行添加书本操作，于是它就执行了响应操作，并把结果写回reply。代码概略如下：

```
case TRANSACTION_addBook: {
    data.enforceInterface(DESCRIPTOR);
    com.lvr.aidldemo.Book _arg0;
    if ((0 != data.readInt())) {
        _arg0 = com.lvr.aidldemo.Book.CREATOR.createFromParcel(d
ata);
    } else {
        _arg0 = null;
    }
    //这里调用服务端实现的addBook方法
    this.addBook(_arg0);
    reply.writeNoException();
    return true;
}
```

然后在 `transact` 方法获得 `_reply` 并返回结果，本例中的addList方法没有返回值。

客户端线程被唤醒。因此调用服务端方法时，应开启子线程，防止**UI**线程堵塞，导致**ANR**。

本文主要介绍**Parcelable**和**Serializable**的作用、效率、区别及选择。

1、作用

Serializable的作用是为了保存对象的属性到本地文件、数据库、网络流、**rmi**以方便数据传输，当然这种传输可以是程序内的也可以是两个程序间的。而Android的Parcelable的设计初衷是因为Serializable效率过慢，为了在程序内不同组件间以及不同**Android**程序间**(AIDL)**高效的传输数据而设计，这些数据仅在内存中存在，Parcelable是通过IBinder通信的消息的载体。

从上面的设计上我们就可以看出优劣了。

2、效率及选择

Parcelable的性能比Serializable好，在内存开销方面较小，所以在内存间数据传输时推荐使用**Parcelable**，如activity间传输数据，而Serializable可将数据持久化方便保存，所以在需要保存或网络传输数据时选择**Serializable**，因为android不同版本Parcelable可能不同，所以不推荐使用Parcelable进行数据持久化。

3、编程实现

对于Serializable，类只需要实现Serializable接口，并提供一个序列化版本id(serialVersionUID)即可。而Parcelable则需要实现writeToParcel、describeContents函数以及静态的CREATOR变量，实际上就是将如何打包和解包的工作自己来定义，而序列化的这些操作完全由底层实现。

Parcelable的一个实现例子如下

```
public class MyParcelable implements Parcelable {
     private int mData;
     private String mStr;

     public int describeContents() {
         return 0;
     }

     // 写数据进行保存
     public void writeToParcel(Parcel out, int flags) {
         out.writeInt(mData);
         out.writeString(mStr);
     }

     // 用来创建自定义的Parcelable的对象
     public static final Parcelable.Creator<MyParcelable> CREATO
R
             = new Parcelable.Creator<MyParcelable>() {
         public MyParcelable createFromParcel(Parcel in) {
             return new MyParcelable(in);
         }

         public MyParcelable[] newArray(int size) {
             return new MyParcelable[size];
         }
     };

     // 读数据进行恢复
     private MyParcelable(Parcel in) {
         mData = in.readInt();
         mStr = in.readString();
     }
 }
```

从上面我们可以看出Parcel的写入和读出顺序是一致的。如果元素是list读出时需要先new一个ArrayList传入，否则会报空指针异常。如下：

```
list = new ArrayList<String>();
in.readStringList(list);
```

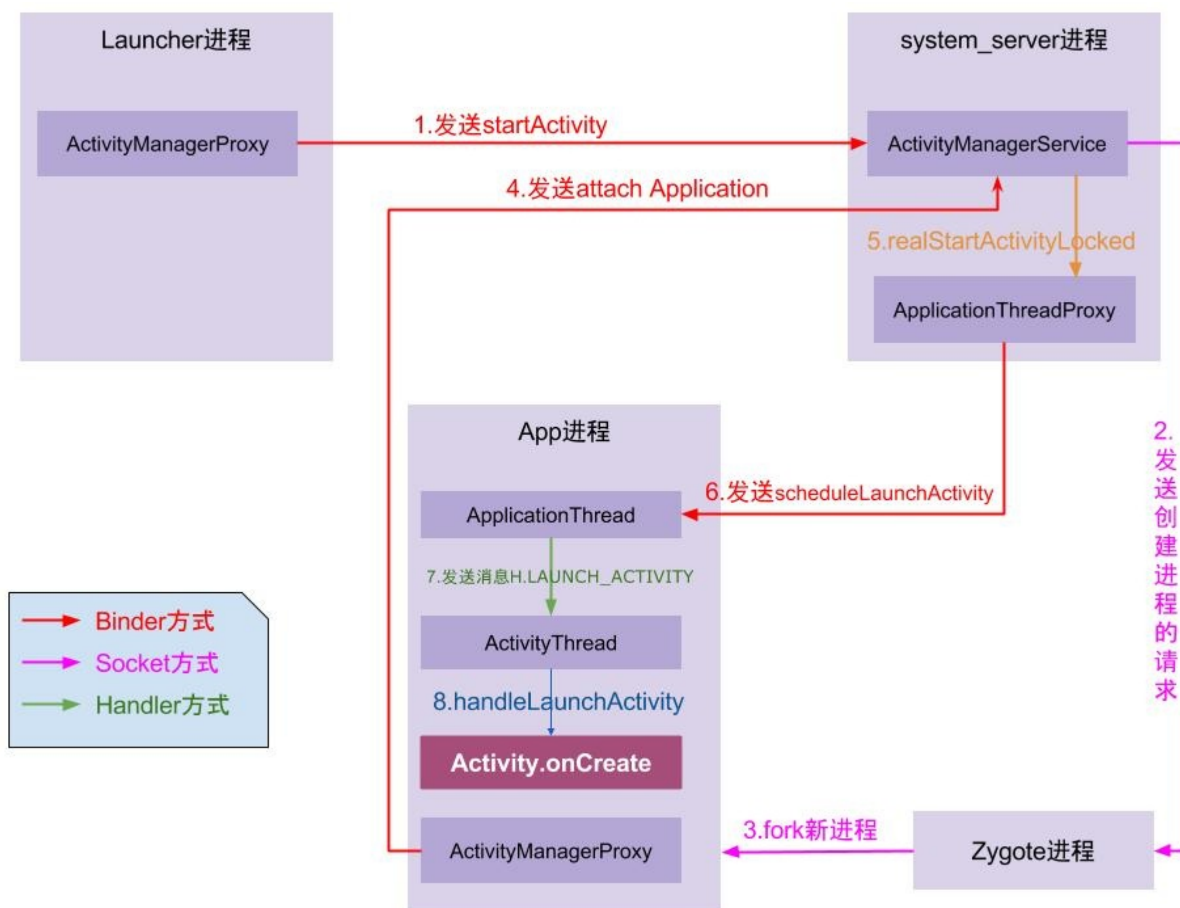
PS: 在自己使用时，read数据时误将前面int数据当作long读出，结果后面的顺序错乱，报如下异常，当类字段较多时务必保持写入和读取的类型及顺序一致。

```
11-21 20:14:10.317: E/AndroidRuntime(21114): Caused by: java.lan
g.RuntimeException: Parcel android.os.Parcel@4126ed60: Unmarshal
ling unknown type code 3014773 at offset 164
```

4、高级功能上

Serializable序列化不保存静态变量，可以使用Transient关键字对部分字段不进行序列化，也可以覆盖writeObject、readObject方法以实现序列化过程自定义。

一、流程概述



启动流程：

①点击桌面App图标，Launcher进程采用Binder IPC向system_server进程发起startActivity请求；

②system_server进程接收到请求后，向zygote进程发送创建进程的请求；

③Zygote进程fork出新的子进程，即App进程；

④App进程，通过Binder IPC向sytem_server进程发起attachApplication请求；

⑤system_server进程在收到请求后，进行一系列准备工作后，再通过binder IPC向App进程发送scheduleLaunchActivity请求；

⑥App进程的binder线程（ApplicationThread）在收到请求后，通过handler向主线程发送LAUNCH_ACTIVITY消息；

⑦主线程在收到Message后，通过发射机制创建目标Activity，并回调Activity.onCreate()等方法。

⑧到此，App便正式启动，开始进入Activity生命周期，执行完onCreate/onStart/onResume方法，UI渲染结束后便可以看到App的主界面。

上面的一些列步骤简单介绍了一个APP启动到主页面显示的过程，可能这些流程中的一些术语看的有些懵，什么是Launcher，什么是zygote，什么是applicationThread.....

下面我们一一介绍。

二、理论基础

1.zygote

zygote意为“受精卵“。Android是基于Linux系统的，而在Linux中，所有的进程都是由init进程直接或者是间接fork出来的，zygote进程也不例外。

在Android系统里面，zygote是一个进程的名字。Android是基于Linux System的，当你的手机开机的时候，Linux的内核加载完成之后就会启动一个叫“init“的进程。在Linux System里面，所有的进程都是由init进程fork出来的，我们的zygote进程也不例外。

我们都知道，每一个App其实都是

- 一个单独的dalvik虚拟机
- 一个单独的进程

所以当系统里面的第一个zygote进程运行之后，在这之后再开启App，就相当于开启一个新的进程。而为了实现资源共用和更快的启动速度，Android系统开启新进程的方式，是通过fork第一个zygote进程实现的。所以说，除了第一个zygote进程，其他应用所在的进程都是zygote的子进程，这下你明白为什么这个进程叫“受精卵”了吧？因为就像是一个受精卵一样，它能快速的分裂，并且产生遗传物质一样的细胞！

2.system_server

SystemServer也是一个进程，而且是由zygote进程fork出来的。

知道了SystemServer的本质，我们对它就不算太陌生了，这个进程是Android Framework里面两大非常重要的进程之一——另外一个进程就是上面的zygote进程。

为什么说SystemServer非常重要呢？因为系统里面重要的服务都是在这个进程里面开启的，比如 ActivityManagerService、PackageManagerService、WindowManagerService等等。

3.ActivityManagerService

ActivityManagerService，简称AMS，服务端对象，负责系统中所有Activity的生命周期。

ActivityManagerService进行初始化的时机很明确，就是在SystemServer进程开启的时候，就会初始化ActivityManagerService。

下面介绍下Android系统里面的服务器和客户端的概念。

其实服务器客户端的概念不仅仅存在于Web开发中，在Android的框架设计中，使用的也是这一种模式。服务器端指的就是所有App共用的系统服务，比如我们这里提到的ActivityManagerService，和前面提到的PackageManagerService、WindowManagerService等等，这些基础的系统服务是被所有的App公用的，当某个App想实现某个操作的时候，要告诉这些系统服务，比如你想打开一个App，那么我们知道了包名和MainActivity类名之后就可以打开

```
Intent intent = new Intent(Intent.ACTION_MAIN);
intent.addCategory(Intent.CATEGORY_LAUNCHER);
ComponentName cn = new ComponentName(packageName, className);

intent.setComponent(cn);
startActivity(intent);
```

但是，我们的App通过调用startActivity()并不能直接打开另外一个App，这个方法会通过一系列的调用，最后还是告诉AMS说："我要打开这个App，我知道他的住址和名字，你帮我打开吧！"所以是AMS来通知zygote进程来fork一个新进程，来开启我们的目标App的。这就像是浏览器想要打开一个超链接一样，浏览器把网页地址发送给服务器，然后还是服务器把需要的资源文件发送给客户端的。

知道了Android Framework的客户端服务器架构之后，我们还需要了解一件事情，那就是我们的App和AMS(SystemServer进程)还有zygote进程分属于三个独立的进程，他们之间如何通信呢？

App与AMS通过Binder进行IPC通信，AMS(SystemServer进程)与zygote通过Socket进行IPC通信。后面具体介绍。

那么AMS有什么用呢？在前面我们知道了，如果想打开一个**App**的话，需要**AMS**去通知**zygote**进程，除此之外，其实所有的**Activity**的开启、暂停、关闭都需要**AMS**来控制，所以我们说，**AMS**负责系统中所有**Activity**的生命周期。

在Android系统中，任何一个**Activity**的启动都是由**AMS**和应用程序进程（主要是**ActivityThread**）相互配合来完成的。**AMS**服务统一调度系统中所有进程的**Activity**启动，而每个**Activity**的启动过程则由其所属的进程具体来完成。

4.Launcher

当我们点击手机桌面上的图标的时候，App就由Launcher开始启动了。但是，你有没有思考过Launcher到底是一个什么东西？

Launcher本质上也是一个应用程序，和我们的**App**一样，也是继承自**Activity**

packages/apps/Launcher2/src/com/android/launcher2/Launcher.java

```
public final class Launcher extends Activity
        implements View.OnClickListener, OnLongClickListener, La
uncherModel.Callbacks,
                   View.OnTouchListener {
                   }
```

Launcher实现了点击、长按等回调接口，来接收用户的输入。既然是普通的App，那么我们的开发经验在这里就仍然适用，比如，我们点击图标的时候，是怎么开启的应用呢？捕捉图标点击事件，然后**startActivity()**发送对应的**Intent**请求呗！是

的，**Launcher**也是这么做的，就是这么**easy**！

5.Instrumentation和ActivityThread

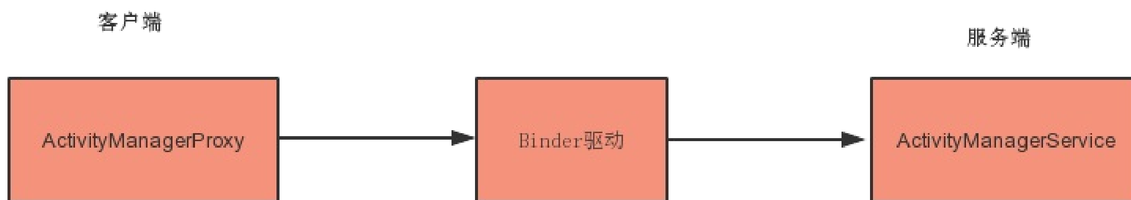
每个Activity都持有Instrumentation对象的一个引用，但是整个进程只会存在一个Instrumentation对象。 Instrumentation这个类里面的方法大多数和Application和Activity有关，这个类就是完成对**Application**和**Activity**初始化和生命周期的工具类。Instrumentation这个类很重要，对Activity生命周期方法的调用根本就离不开他，他可以说是一个大管家。

ActivityThread，依赖于UI线程。App和AMS是通过Binder传递信息的，那么ActivityThread就是专门与AMS的外交工作的。

6.ApplicationThread

前面我们已经知道了App的启动以及Activity的显示都需要AMS的控制，那么我们便需要和服务端的沟通，而这个沟通是双向的。

客户端**-->**服务端



而且由于继承了同样的公共接口类，ActivityManagerProxy提供了与ActivityManagerService一样的函数原型，使用户感觉不出Server是运行在本地还是远端，从而可以更加方便的调用这些重要的系统服务。

服务端**-->**客户端

还是通过Binder通信，不过是换了另外一对，换成了ApplicationThread和ApplicationThreadProxy。



他们也都实现了相同的接口IApplicationThread

```
  private class ApplicationThread extends ApplicationThreadNative
 {}

  public abstract class ApplicationThreadNative extends Binder implements IApplicationThread{}

  class ApplicationThreadProxy implements IApplicationThread {}
```

好了，前面罗里吧嗦的一大堆，介绍了一堆名词，可能不太清楚，没关系，下面结合流程图介绍。

三、启动流程

1.创建进程

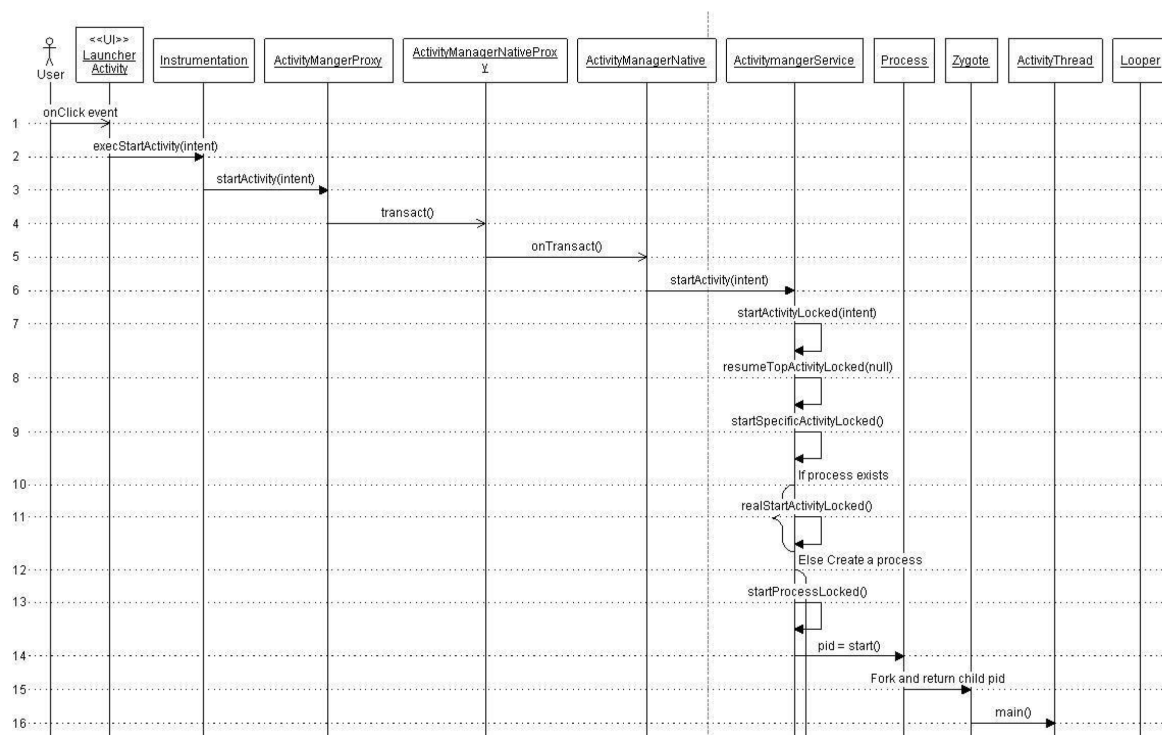
①先从Launcher的startActivity()方法，通过Binder通信，调用ActivityManagerService的startActivity方法。

②一系列折腾，最后调用startProcessLocked()方法来创建新的进程。

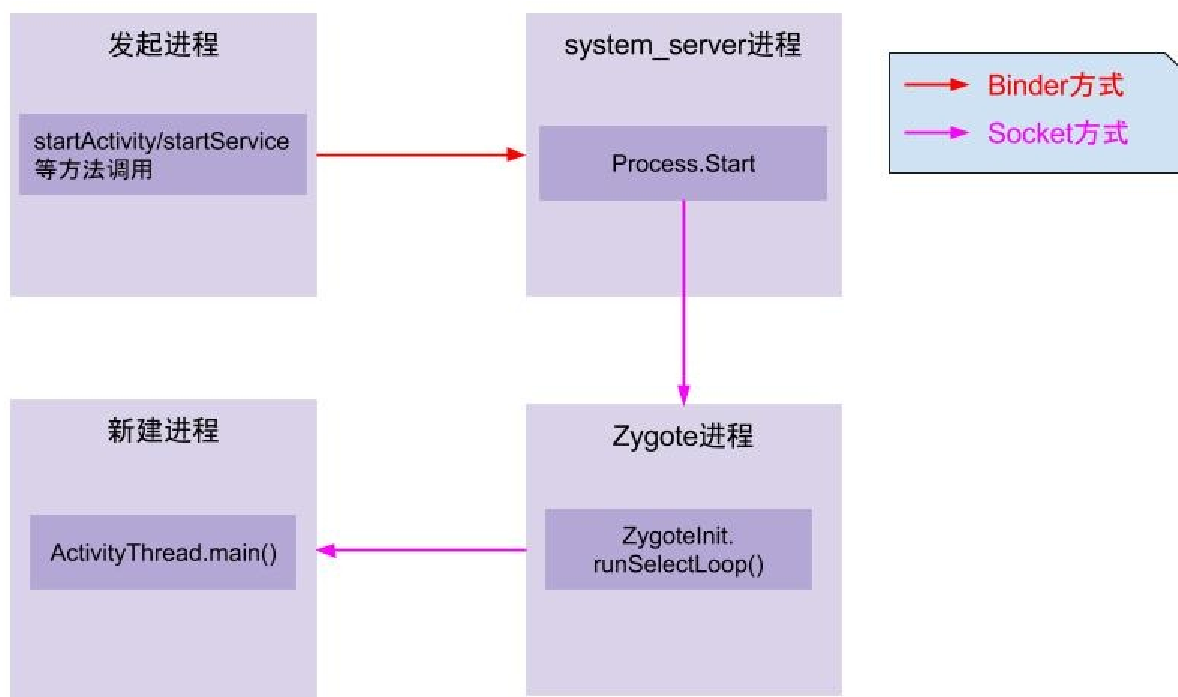
③该方法会通过前面讲到的socket通道传递参数给Zygote进程。Zygote孵化自身。调用ZygoteInit.main()方法来实例化ActivityThread对象并最终返回新进程的pid。

④调用ActivityThread.main()方法，ActivityThread随后依次调用Looper.prepareLoop()和Looper.loop()来开启消息循环。

方法调用流程图如下:



更直白的流程解释：



①App发起进程：当从桌面启动应用，则发起进程便是Launcher所在进程；当从某App内启动远程进程，则发送进程便是该App所在进程。发起进程先通过binder发送消息给system_server进程；

②system_server进程：调用Process.start()方法，通过socket向zygote进程发送创建新进程的请求；

③zygote进程：在执行ZygoteInit.main()后便进入runSelectLoop()循环体内，当有客户端连接时便会执行ZygoteConnection.runOnce()方法，再经过层层调用后fork出新的应用进程；

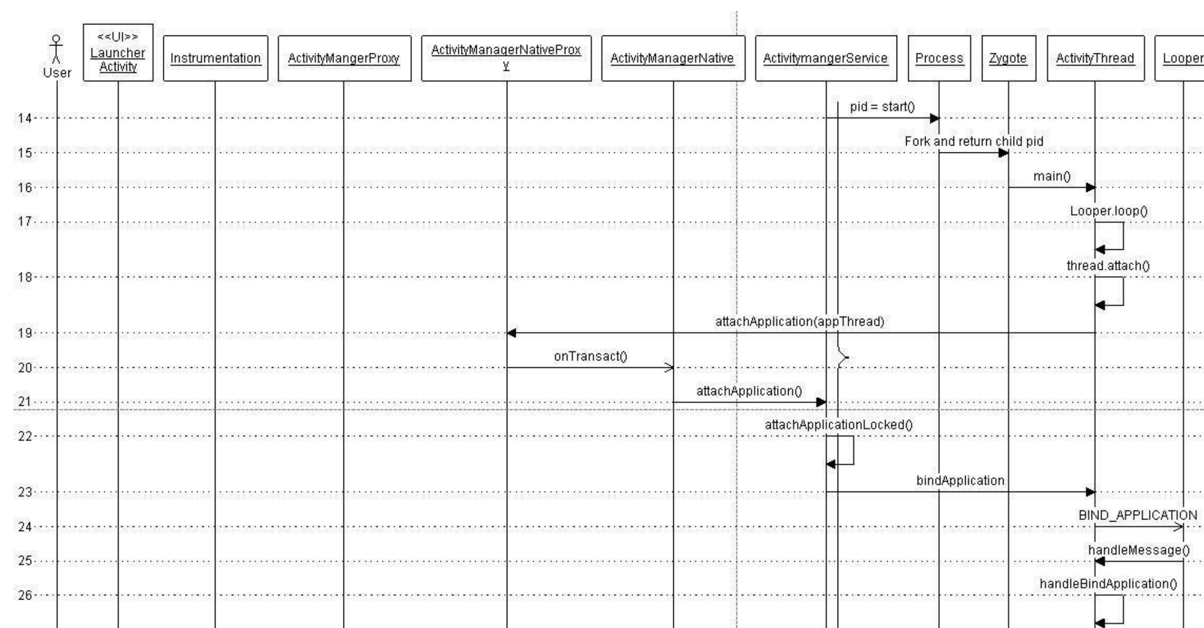
④新进程：执行handleChildProc方法，最后调用ActivityThread.main()方法。

2.绑定Application

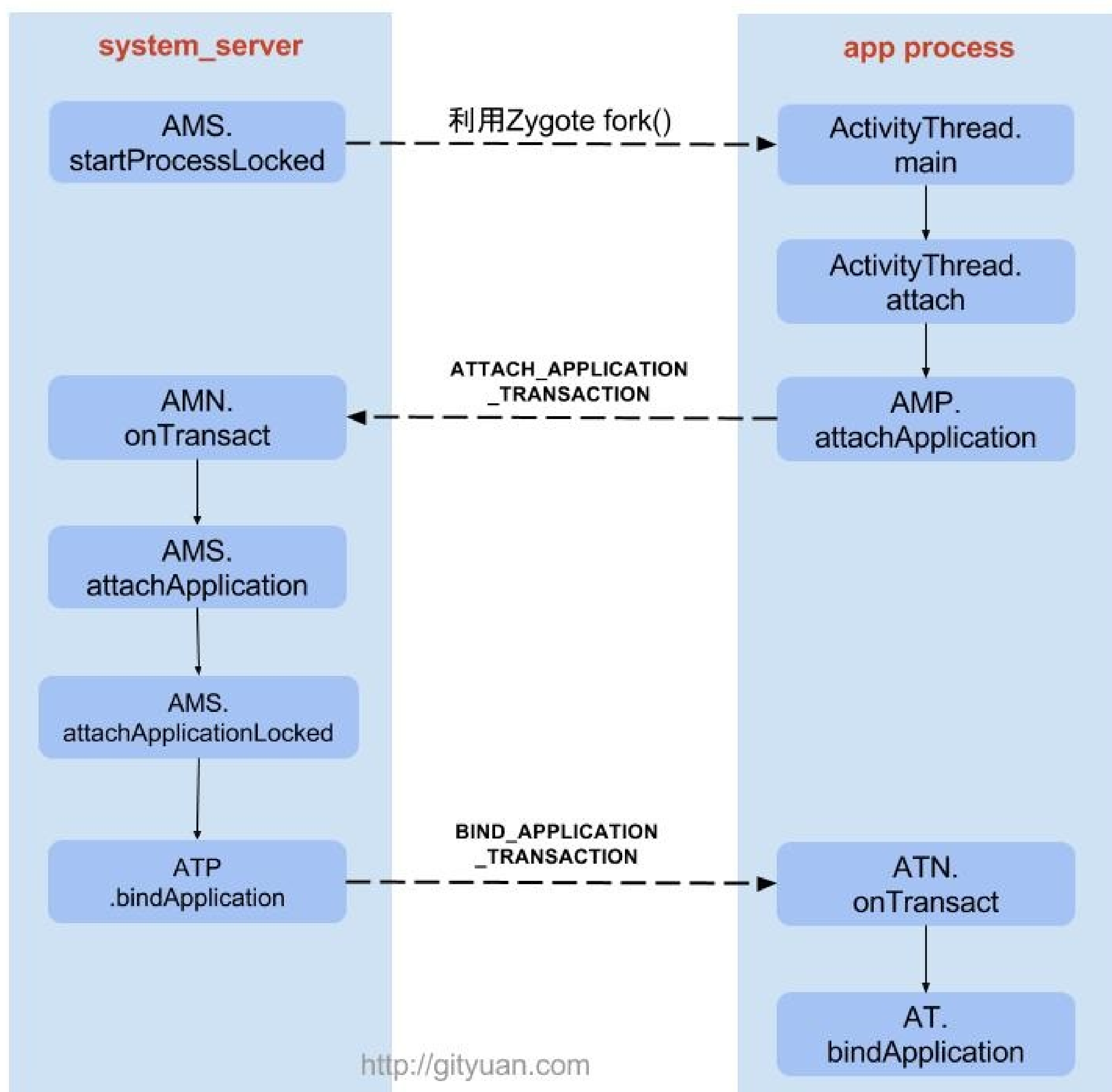
上面创建进程后，执行ActivityThread.main()方法，随后调用attach()方法。

将进程和指定的Application绑定起来。这个是通过上节的ActivityThread对象中调用bindApplication()方法完成的。该方法发送一个BIND_APPLICATION的消息到消息队列中, 最终通过handleBindApplication()方法处理该消息. 然后调用makeApplication()方法来加载App的classes到内存中。

方法调用流程图如下：



更直白的流程解释：

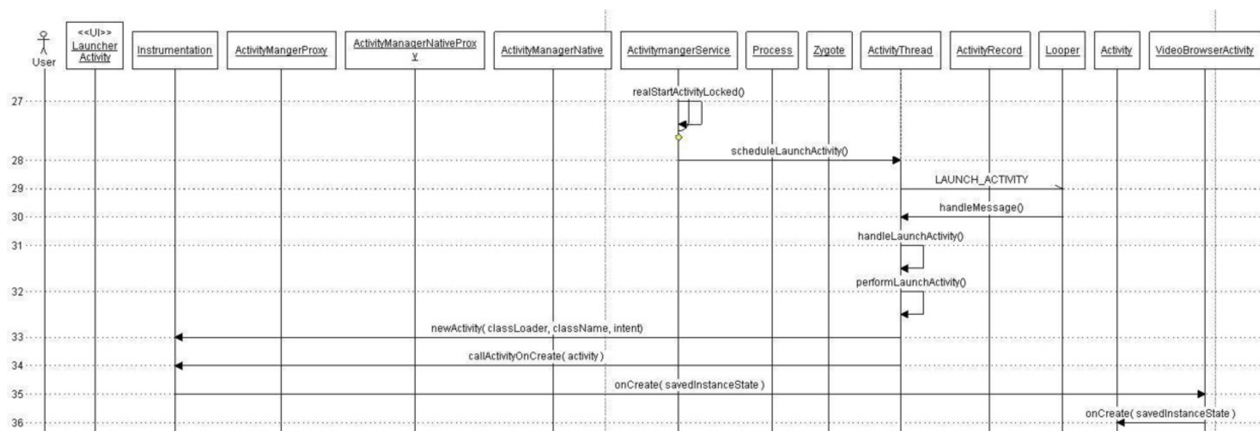


（如果看不懂AMS,ATP等名词，后面有解释）

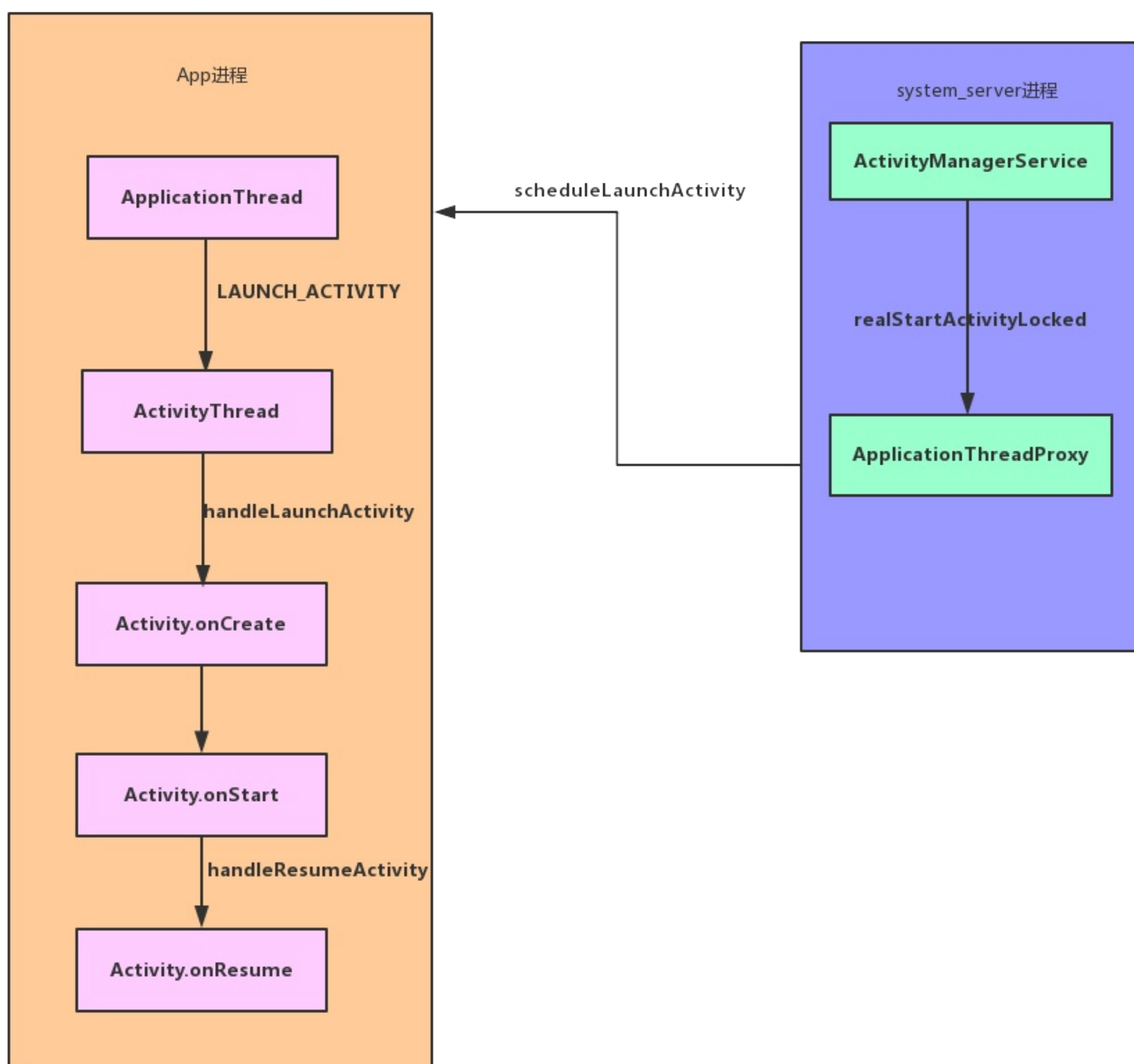
3.显示Activity界面

经过前两个步骤之后, 系统已经拥有了该application的进程。 后面的调用顺序就是普通的从一个已经存在的进程中启动一个新进程的activity了。

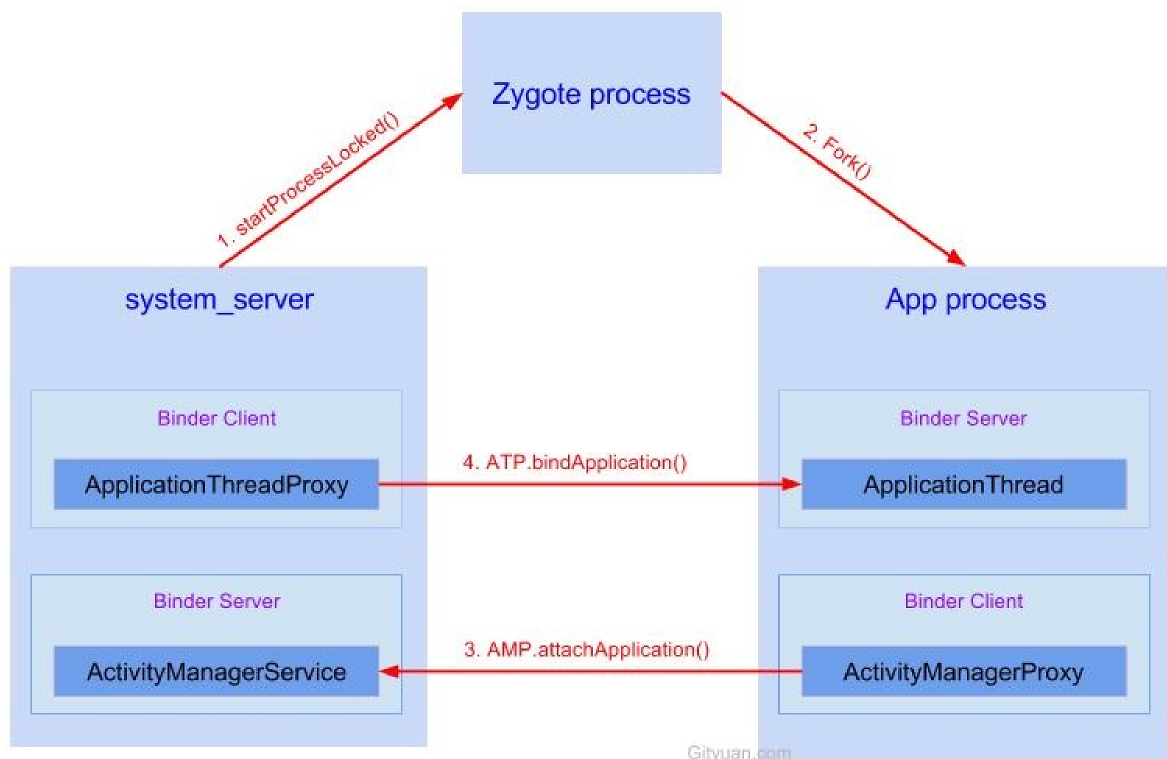
实际调用方法是realStartActivity(), 它会调用application线程对象中的scheduleLaunchActivity()发送一个LAUNCH_ACTIVITY消息到消息队列中, 通过handleLaunchActivity()来处理该消息。在 handleLaunchActivity()通过performLaunchActiivty()方法回调Activity的onCreate()方法和onStart()方法，然后通过handleResumeActivity()方法，回调Activity的onResume()方法，最终显示Activity界面。



更直白的流程解释：



四、Binder通信



简称:

ATP: ApplicationThreadProxy

AT: ApplicationThread

AMP: ActivityManagerProxy

AMS: ActivityManagerService

图解:

①system_server进程中调用startProcessLocked方法,该方法最终通过socket方式,将需要创建新进程的消息告知Zygote进程,并阻塞等待Socket返回新创建进程的pid;

②Zygote进程接收到system_server发送过来的消息, 则通过fork的方法，将zygote自身进程复制生成新的进程，并将ActivityThread相关的资源加载到新进程app process,这个进程可能是用于承载activity等组件;

③ 在新进程app process向servicemanager查询system_server进程中binder服务端AMS, 获取相对应的Client端,也就是AMP. 有了这一对binder c/s对, 那么app process便可以通过binder向跨进程system_server发送请求,即attachApplication()

④system_server进程接收到相应binder操作后,经过多次调用,利用ATP向app process发送binder请求, 即bindApplication. system_server拥有ATP/AMS, 每一个新创建的进程都会有一个相应的AT/AMP,从而可以跨进程 进行相互通信. 这便是进程创建过程的完整生态链。

以上大概介绍了一个APP从启动到主页面显示经历的流程，主要从宏观角度介绍了其过程，具体可结合源码理解。

一、Android性能优化的方面

针对Android的性能优化，主要有以下几个有效的优化方法：

1.布局优化

2.绘制优化

3.内存泄漏优化

4.响应速度优化

5.ListView/RecycleView及Bitmap优化

6.线程优化

7.其他性能优化的建议

下面我们具体来介绍关于以上这几个方面优化的具体思路及解决方案。

二、布局优化

关于布局优化的思想很简单，就是尽量减少布局文件的层级。这个道理很浅显，布局中的层级少了，就意味着Android绘制时的工作量少了，那么程序的性能自然就提高了。

如何进行布局优化？

①删除布局中无用的控件和层次，其次有选择地使用性能比较低的**ViewGroup**。

关于有选择地使用性能比较低的ViewGroup,这就需要我们开发就实际灵活选择了。

例如：如果布局中既可以使用LinearLayout也可以使用RelativeLayout，那么就采用LinearLayout，这是因为RelativeLayout的功能比较复杂，它的布局过程需要花费更多的CPU时间。FrameLayout和LinearLayout一样都是一种简单高效的ViewGroup，因此可以考虑使用它们，但是很多时候单纯通过一个**LinearLayout**或者**FrameLayout**无法实现产品效果，需要通过嵌套的方式来完成。这种情况下还是建议采用**RelativeLayout,**因为**ViewGroup**的嵌套就相当于增加了布局的层级，同样会降低程序的性能。

②采用标签,标签,**ViewStub**。

标签主要用于布局重用。

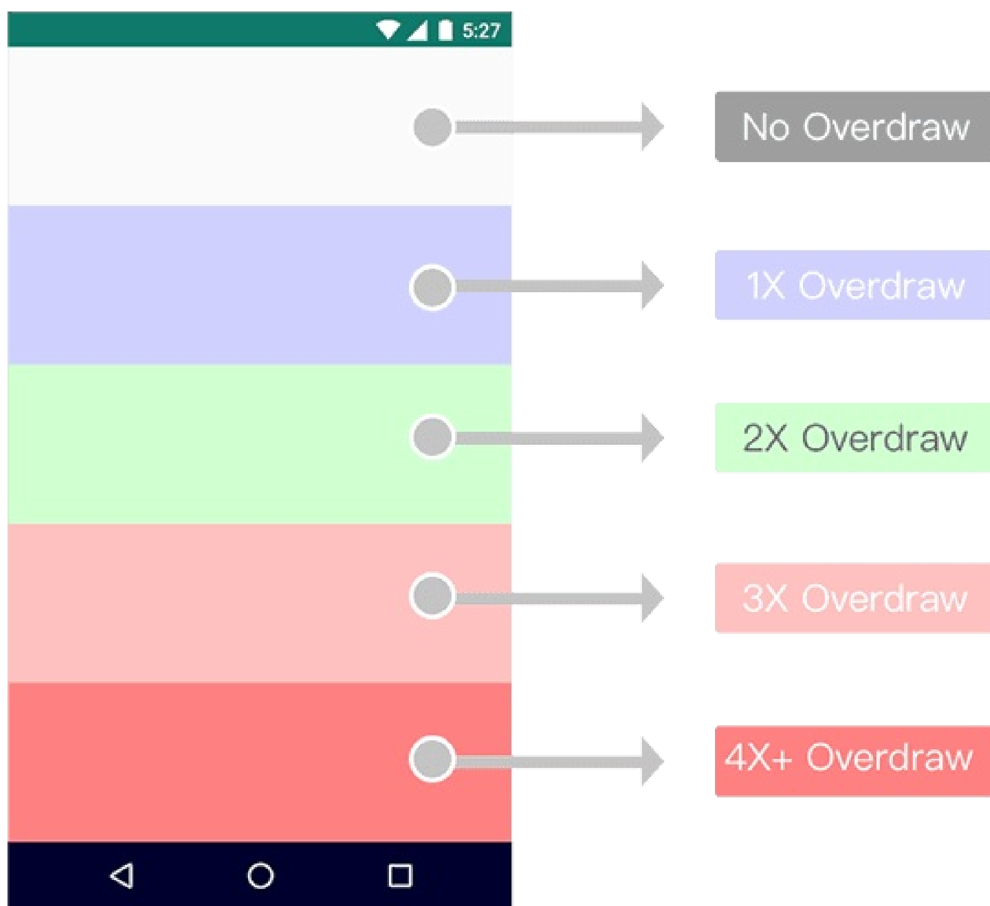
标签一般和配合使用，可以降低减少布局的层级。

ViewStub提供了按需加载的功能，当需要时才会将ViewStub中的布局加载到内存，提高了程序初始化效率。

③避免多度绘制

过度绘制（Overdraw）描述的是屏幕上的某个像素在同一帧的时间内被绘制了多次。在多层次重叠的 UI 结构里面，如果不可见的 UI 也在做绘制的操作，会导致某些像素区域被绘制了多次，同时也会浪费大量的 CPU 以及 GPU 资源。

如下所示，有些部分在布局时，会被重复绘制。



关于过度绘制产生的一般场景及解决方案，参考：Android 过度绘制优化

三、绘制优化

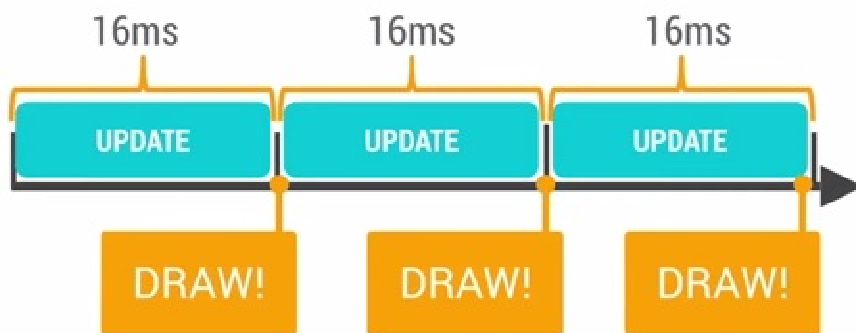
绘制优化是指**View**的**onDraw**方法要避免执行大量的操作，这主要体现在两个方面：

①onDraw中不要创建新的局部对象。

因为onDraw方法可能会被频繁调用，这样就会在一瞬间产生大量的临时对象，这不仅占用了过多的内存而且还会导致系统更加频繁gc，降低了程序的执行效率。

②onDraw方法中不要做耗时的任务，也不能执行成千上万次的循环操作，尽管每次循环都很轻量级，但是大量的循环仍然十分抢占**CPU**的时间片，这会造成**View**的绘制过程不流畅。

按照Google官方给出的性能优化典范中的标准，View的绘制频率保证60fps是最佳的，这就要求每帧绘制时间不超过16ms(16ms = 1000/60)，虽然程序很难保证16ms这个时间，但是尽量降低onDraw方法中的复杂度总是切实有效的。



四、内存泄漏优化

内存泄漏是开发过程中的一个需要重视的问题，但是由于内存泄露问题对开发人员的经验和开发意识有较高的要求，因此也是开发人员最容易犯的错误之一。

内存泄露的优化分为两个方面：

①在开发过程中避免写出有内存泄漏的代码

②通过一些分析工具比如**MAT**来找出潜在的内存泄露，然后解决。

对应于两种不同情况，一个是了解内存泄漏的可能场景以及如何规避，二是怎么查找内存泄漏。

1.那么我们就先了解什么是内存泄漏?这样我们才能知道如何避免。

大家都知道，java是有垃圾回收机制的，这使得java程序员比C++程序员轻松了许多，存储申请了，不用心心念念要加一句释放，java虚拟机会派出一些回收线程兢兢业业不定时地回收那些不再被需要的内存空间（注意回收的不是对象本身，而是对象占据的内存空间）。

Q1：什么叫不再被需要的内存空间？

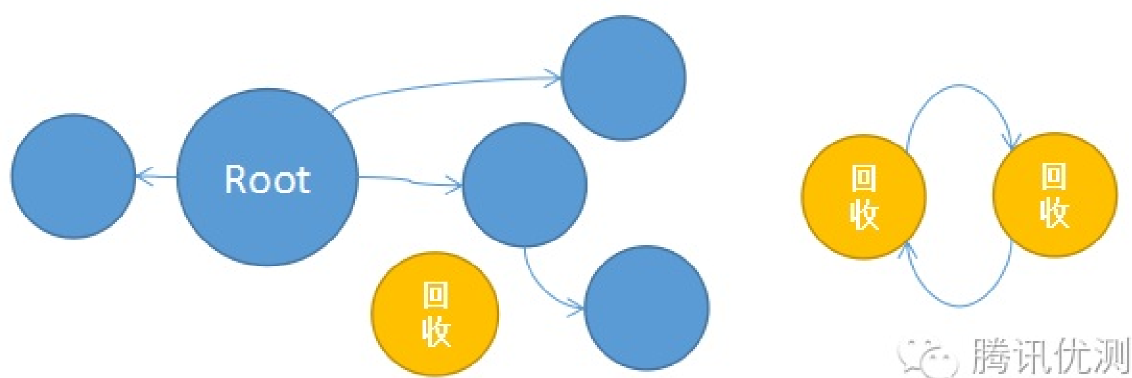
答：Java没有指针，全凭引用来和对象进行关联，通过引用来操作对象。如果一个对象没有与任何引用关联，那么这个对象也就不太可能被使用到了，回收器便是把这些“无任何引用的对象”作为目标，回收了它们占据的内存空间。

Q2：如何分辨为对象无引用？

答：2种方法

引用计数法直接计数，简单高效，Python便是采用该方法。但是如果出现 两个对象相互引用，即使它们都无法被外界访问到，计数器不为0它们也始终不会被回收。为了解决该问题，java采用的是b方法。

可达性分析法这个方法设置了一系列的“GC Roots”对象作为索引起点，如果一个对象 与起点对象之间均无可达路径，那么这个不可达的对象就会成为回收对象。这种方法处理 两个对象相互引用的问题，如果两个对象均没有外部引用，会被判断为不可达对象进而被回收（如下图）。

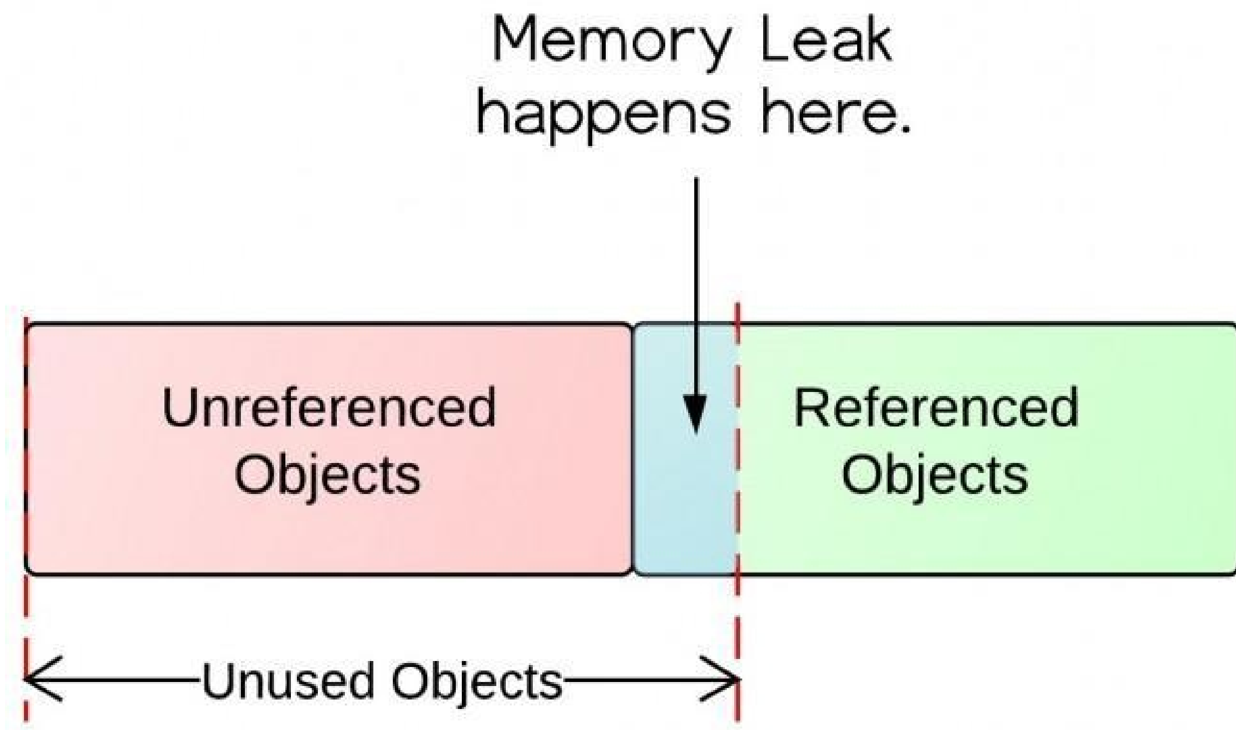


Q3：有了回收机制，放心大胆用不会有内存泄漏？

答：答案当然是No！

虽然垃圾回收器会帮我们干掉大部分无用的内存空间，但是对于还保持着引用，但逻辑上已经不会再用到的对象，垃圾回收器不会回收它们。这些对象积累在内存中，直到程序结束，就是我们所说的“内存泄漏”。 当然了，用户对单次的内存泄漏并没有什么感知，但当泄漏积累到内存都被消耗完，就会导致卡顿，崩溃。

下面这张图可以帮助我们更好地理解对象的状态，以及内存泄漏的情况



左边未引用的对象是会被GC回收的，右边被引用的对象不会被GC回收，但是未使用的对象中除了未引用的对象，还包括已被引用的一部分对象，那么内存泄漏久发生这部分已被引用但未使用的对象。

2.Android一般在什么情况下会出现内存泄漏？

①集合类泄漏 ②单例/静态变量造成的内存泄漏 ③匿名内部类/非静态内部类 ④资源未关闭造成的内存泄漏

大概可以分为以上几类，还有一些经常会听到的Hanlder,AsyncTask引起内存泄漏，都属于上述③中的情况。

那么上述四种情况是怎么造成的内存泄漏，具体是什么原因，以及Android中一些知名的引起内存泄漏的原因，以及解决方法是怎么样的？

3.Android怎么分析内存泄漏？

上面介绍了内存泄漏的场景，对应的有一些解决方案。

那么在内存泄漏已经发生的情况下，我们该如何解决呢？

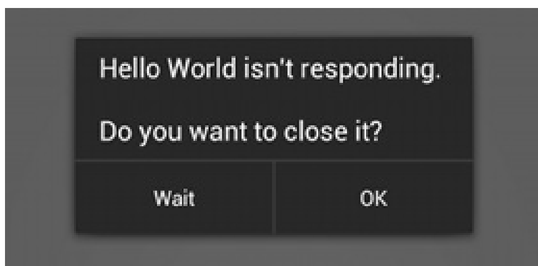
我们可以通过MAT(Memory Analyzer Tool)，或者 LeakCanary来检测Android中的内存泄漏。

五、响应速度优化

响应速度优化的核心思想就是避免在主线程中做耗时操作。

如果有耗时操作，可以开启子线程执行，即采用异步的方式来执行耗时操作。

如果在主线程中做太多事情，会导致Activity启动时出现黑屏现象，甚至ANR。

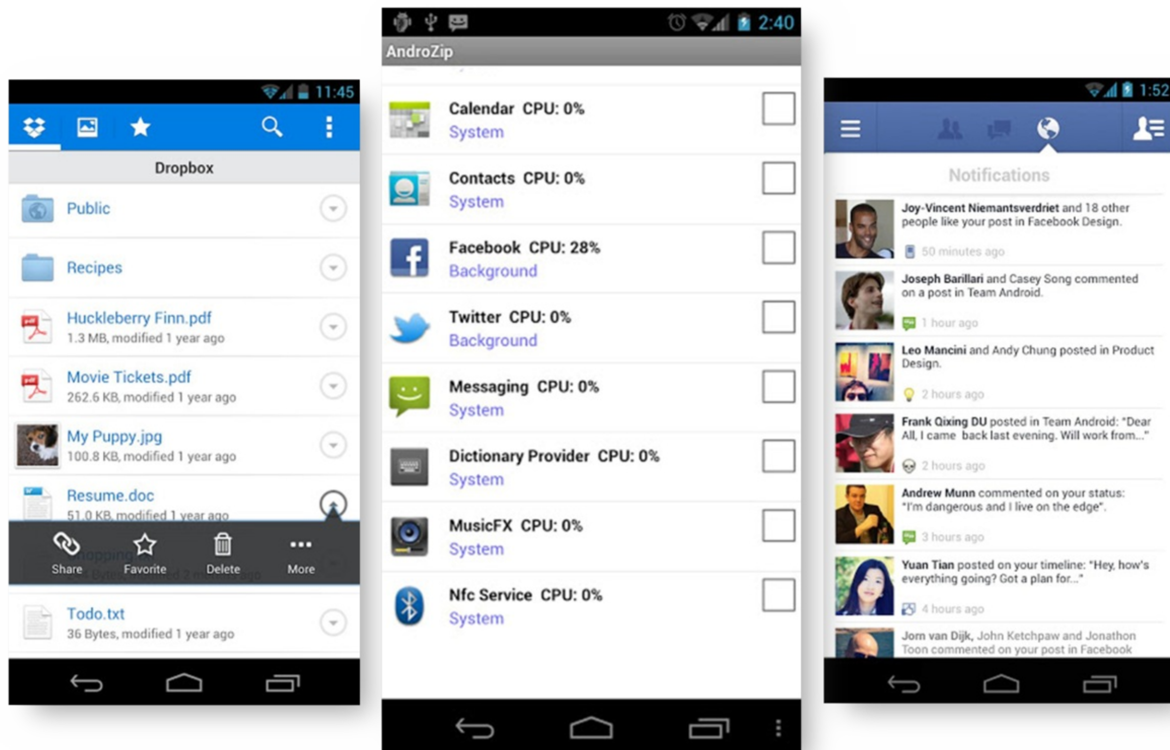


Android规定，**Activity**如果**5**秒钟之内无法响应屏幕触摸事件或者键盘输入事件就会出现**ANR**，而**BroadcastReceiver**如果**10**秒钟之内还未执行完操作也会出现**ANR**。

为了避免ANR，可以开启子线程执行耗时操作，但是子线程不能更新UI，所以需要子线程与主线程进行通信来解决子线程执行耗时任务后，通知主线程更新UI的场景。关于这部分，需要掌握Handler消息机制，AsyncTask，IntentService等内容。

然而，在实际开发中，ANR仍然不可避免的发生了，而且很难从代码上发现，这时候就要用到ANR日志分析。当一个进程发生了ANR之后，系统会在/data/anr目录下创建一个文件traces.txt，通过分析这个文件就能定位出ANR的原因。

六、ListView/RecycleView及Bitmap优化



ListView/RecycleView的优化思想主要从以下几个方面入手：

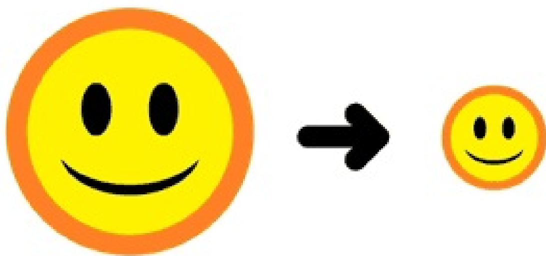
①使用ViewHolder模式来提高效率

②异步加载：耗时的操作放在异步线程中

③ListView/RecycleView的滑动时停止加载和分页加载

具体优化建议及详情，参考：ListView的优化

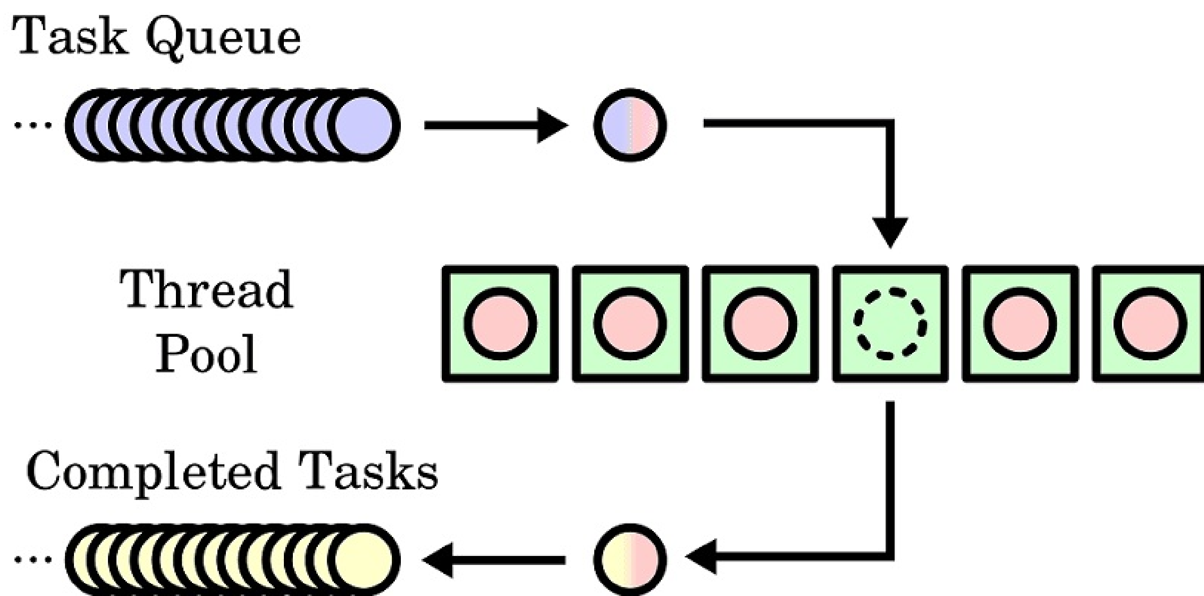
Bitmap优化



主要是对加载图片进行压缩，避免加载图片多大导致OOM出现。

七、线程优化

线程优化的思想就是采用线程池，避免程序中存在大量的**Thread**。线程池可以重用内部的线程，从而避免了线程的创建和销毁锁带来的性能开销，同时线程池还能有效地控制线程池的最大并法术，避免大量的线程因互相抢占系统资源从而导致阻塞现象的发生。因此在实际开发中，尽量采用线程池，而不是每次都要创建一个Thread对象。



八、其他性能优化建议

①避免过度的创建对象

②不要过度使用枚举，枚举占用的内存空间要比整型大

③常量请使用static final来修饰

④使用一些Android特有的数据结构，比如SparseArray和Pair等

⑤适当采用软引用和弱引用

⑥采用内存缓存和磁盘缓存

⑦尽量采用静态内部类，这样可以避免潜在的由于内部类而导致的内存泄漏。

以上是关于Android性能优化方面，我们一些入手点。从这些方面，我们可以在平时的开发中注意，避免类似错误，提高Android程序的性能，但是其中一些方面的要求则需要我们不断的学习，以及平时良好的意识与习惯。由于自己开发经验几乎

为0，没办法根据实际经验来说明，只能写下这篇文章来提醒自己以后开发的时候需要注意和培养的地方。

Android 内存泄漏总结

内存管理的目的就是让我们在开发中怎么有效的避免我们的应用出现内存泄漏的问题。内存泄漏大家都不陌生了，简单粗俗的讲，就是该被释放的对象没有释放，一直被某个或某些实例所持有却不再被使用导致 GC 不能回收

我会从 java 内存泄漏的基础知识开始，并通过具体例子来说明 Android 引起内存泄漏的各种原因，以及如何利用工具来分析应用内存泄漏，最后再做总结。

篇幅有些长，大家可以分几节来看！

Java 内存分配策略

Java 程序运行时的内存分配策略有三种,分别是静态分配,栈式分配,和堆式分配，对应的，三种存储策略使用的内存空间主要分别是静态存储区（也称方法区）、栈区和堆区。

- 静态存储区（方法区）：主要存放静态数据、全局 static 数据和常量。这块内存在程序编译时就已经分配好，并且在程序整个运行期间都存在。
- 栈区 ：当方法被执行时，方法体内的局部变量都在栈上创建，并在方法执行结束时这些局部变量所持有的内存将会自动被释放。因为栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。
- 堆区 ： 又称动态内存分配，通常就是指在程序运行时直接 new 出来的内存。这部分内存在不使用时将会由 Java 垃圾回收器来负责回收。

栈与堆的区别：

在方法体内定义的（局部变量）一些基本类型的变量和对象的引用变量都是在方法的栈内存中分配的。当在一段方法块中定义一个变量时，Java 就会在栈中为该变量分配内存空间，当超过该变量的作用域后，该变量也就无效了，分配给它的内存空间也将被释放掉，该内存空间可以被重新使用。

堆内存用来存放所有由 new 创建的对象（包括该对象其中的所有成员变量）和数组。在堆中分配的内存，将由 Java 垃圾回收器来自动管理。在堆中产生了一个数组或者对象后，还可以在栈中定义一个特殊的变量，这个变量的取值等于数组或者

对象在堆内存中的首地址，这个特殊的变量就是我们上面说的引用变量。我们可以通过这个引用变量来访问堆中的对象或者数组。

举个例子:

```
public class Sample() {
    int s1 = 0;
    Sample mSample1 = new Sample();

    public void method() {
        int s2 = 1;
        Sample mSample2 = new Sample();
    }
}

Sample mSample3 = new Sample();
```

Sample 类的局部变量 s2 和引用变量 mSample2 都是存在于栈中，但 mSample2 指向的对象是存在于堆上的。 mSample3 指向的对象实体存放在堆上，包括这个对象的所有成员变量 s1 和 mSample1，而它自己存在于栈中。

结论：

局部变量的基本数据类型和引用存储于栈中，引用的对象实体存储于堆中。—— 因为它们属于方法中的变量，生命周期随方法而结束。

成员变量全部存储于堆中（包括基本数据类型，引用和引用的对象实体）—— 因为它们属于类，类对象终究是要被new出来使用的。

了解了 Java 的内存分配之后，我们再来看看 Java 是怎么管理内存的。

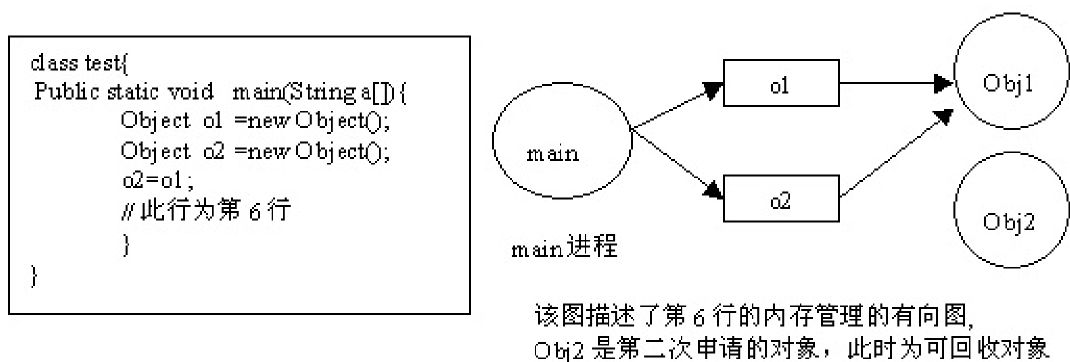
Java是如何管理内存

Java的内存管理就是对象的分配和释放问题。在 Java 中，程序员需要通过关键字 new 为每个对象申请内存空间 (基本类型除外)，所有的对象都在堆 (Heap)中分配空间。另外，对象的释放是由 GC 决定和执行的。在 Java 中，内存的分配是由程序完成的，而内存的释放是由 GC 完成的，这种收支两条线的方法确实简化了程序员

的工作。但同时，它也加重了JVM的工作。这也是 Java 程序运行速度较慢的原因之一。因为，GC 为了能够正确释放对象，GC 必须监控每一个对象的运行状态，包括对象的申请、引用、被引用、赋值等，GC 都需要进行监控。

监视对象状态是为了更加准确地、及时地释放对象，而释放对象的根本原则就是该对象不再被引用。

为了更好理解 GC 的工作原理，我们可以将对象考虑为有向图的顶点，将引用关系考虑为图的有向边，有向边从引用者指向被引对象。另外，每个线程对象可以作为一个图的起始顶点，例如大多程序从 main 进程开始执行，那么该图就是以 main 进程顶点开始的一棵根树。在这个有向图中，根顶点可达的对象都是有效对象，GC将不回收这些对象。如果某个对象 (连通子图)与这个根顶点不可达(注意，该图为有向图)，那么我们认为这个(这些)对象不再被引用，可以被 GC 回收。 以下，我们举一个例子说明如何用有向图表示内存管理。对于程序的每一个时刻，我们都有一个有向图表示JVM的内存分配情况。以下右图，就是左边程序运行到第6行的示意图。



Java使用有向图的方式进行内存管理，可以消除引用循环的问题，例如有三个对象，相互引用，只要它们和根进程不可达的，那么GC也是可以回收它们的。这种方式的优点是管理内存的精度很高，但是效率较低。另外一种常用的内存管理技术是使用计数器，例如COM模型采用计数器方式管理构件，它与有向图相比，精度行低(很难处理循环引用的问题)，但执行效率很高。

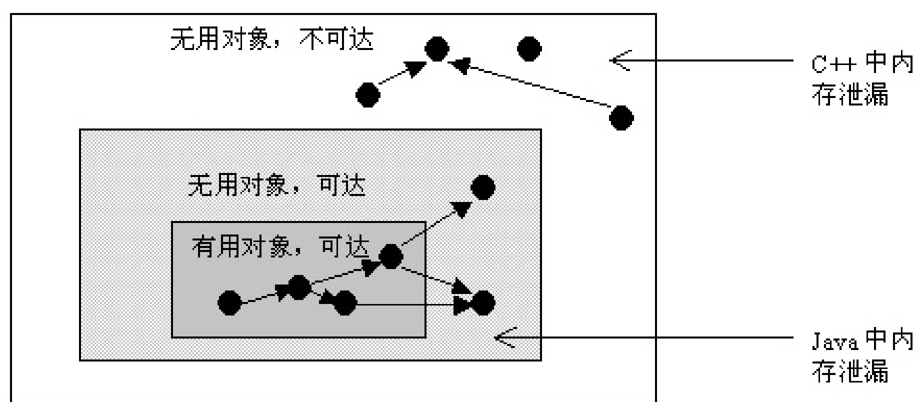
什么是Java中的内存泄露

在Java中，内存泄漏就是存在一些被分配的对象，这些对象有下面两个特点，首先，这些对象是可达的，即在有向图中，存在通路可以与其相连；其次，这些对象是无用的，即程序以后不会再使用这些对象。如果对象满足这两个条件，这些对象

就可以判定为Java中的内存泄漏，这些对象不会被GC所回收，然而它却占用内存。

在C++中，内存泄漏的范围更大一些。有些对象被分配了内存空间，然后却不可达，由于C++中没有GC，这些内存将永远收不回来。在Java中，这些不可达的对象都由GC负责回收，因此程序员不需要考虑这部分的内存泄露。

通过分析，我们得知，对于C++，程序员需要自己管理边和顶点，而对于Java程序员只需要管理边就可以了(不需要管理顶点的释放)。通过这种方式，Java提高了编程的效率。



因此，通过以上分析，我们知道在Java中也有内存泄漏，但范围比C++要小一些。因为Java从语言上保证，任何对象都是可达的，所有的不可达对象都由GC管理。

对于程序员来说，GC基本是透明的，不可见的。虽然，我们只有几个函数可以访问GC，例如运行GC的函数System.gc()，但是根据Java语言规范定义， 该函数不保证JVM的垃圾收集器一定会执行。因为，不同的JVM实现者可能使用不同的算法管理GC。通常，GC的线程的优先级别较低。JVM调用GC的策略也有很多种，有的是内存使用到达一定程度时，GC才开始工作，也有定时执行的，有的是平缓执行GC，有的是中断式执行GC。但通常来说，我们不需要关心这些。除非在一些特定的场合，GC的执行影响应用程序的性能，例如对于基于Web的实时系统，如网络游戏等，用户不希望GC突然中断应用程序执行而进行垃圾回收，那么我们需要调整GC的参数，让GC能够通过平缓的方式释放内存，例如将垃圾回收分解为一系列的小步骤执行，Sun提供的HotSpot JVM就支持这一特性。

同样给出一个 Java 内存泄漏的典型例子，

```
Vector v = new Vector(10);
for (int i = 1; i < 100; i++) {
    Object o = new Object();
    v.add(o);
    o = null;
}
```

在这个例子中，我们循环申请Object对象，并将所申请的对象放入一个 Vector 中，如果我们仅仅释放引用本身，那么 Vector 仍然引用该对象，所以这个对象对 GC 来说是不可回收的。因此，如果对象加入到Vector 后，还必须从 Vector 中删除，最简单的方法就是将 Vector 对象设置为 null。

Android中常见的内存泄漏汇总

- 集合类泄漏

  集合类如果仅仅有添加元素的方法，而没有相应的删除机制，导致内存被占用。如果这个集合类是全局性的变量 (比如类中的静态属性，全局性的 map 等即有静态引用或 final 一直指向它)，那么没有相应的删除机制，很可能导致集合所占用的内存只增不减。比如上面的典型例子就是其中一种情况，当然实际上我们在项目中肯定不会写这么 2B 的代码，但稍不注意还是很容易出现这种情况，比如我们都喜欢通过 HashMap 做一些缓存之类的事，这种情况就要多留一些心眼。

- 单例造成的内存泄漏

  由于单例的静态特性使得其生命周期跟应用的生命周期一样长，所以如果使用不恰当的话，很容易造成内存泄漏。比如下面一个典型的例子，

```
public class AppManager {
  private static AppManager instance;
  private Context context;
  private AppManager(Context context) {
  this.context = context;
  }
  public static AppManager getInstance(Context context) {
    if (instance == null) {
    instance = new AppManager(context);
    }
    return instance;
  }
}
```

这是一个普通的单例模式，当创建这个单例的时候，由于需要传入一个Context，所以这个Context的生命周期的长短至关重要：

1、如果此时传入的是 Application 的 Context，因为 Application 的生命周期就是整个应用的生命周期，所以这将没有任何问题。

2、如果此时传入的是 Activity 的 Context，当这个 Context 所对应的 Activity 退出时，由于该 Context 的引用被单例对象所持有，其生命周期等于整个应用程序的生命周期，所以当前 Activity 退出时它的内存并不会被回收，这就造成泄漏了。

正确的方式应该改为下面这种方式：

```
public class AppManager {
  private static AppManager instance;
  private Context context;
  private AppManager(Context context) {
    this.context = context.getApplicationContext();// 使用Applica
tion 的context
 }
  public static AppManager getInstance(Context context) {
    if (instance == null) {
      instance = new AppManager(context);
    }
    return instance;
  }
}
```

或者这样写，连 Context 都不用传进来了：

```
在你的 Application 中添加一个静态方法，getContext() 返回 Application
的 context，

...

context = getApplicationContext();

...
   /**
     * 获取全局的context
     * @return 返回全局context对象
     */
    public static Context getContext(){
        return context;
    }

public class AppManager {
  private static AppManager instance;
  private Context context;
  private AppManager() {
    this.context = MyApplication.getContext();// 使用Application
的context
  }
  public static AppManager getInstance() {
    if (instance == null) {
      instance = new AppManager();
    }
    return instance;
  }
}
```

- 匿名内部类/非静态内部类和异步线程
    - 非静态内部类创建静态实例造成的内存泄漏

      有的时候我们可能会在启动频繁的Activity中，为了避免重复创建相同的数据资源，可能会出现这种写法：

```
public class MainActivity extends AppCompatActivity {
  private static TestResource mResource = null;
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    if(mManager == null){
      mManager = new TestResource();
    }
    //...
    }
  class TestResource {
  //...
  }
}
```

这样就在Activity内部创建了一个非静态内部类的单例，每次启动Activity时都会使用该单例的数据，这样虽然避免了资源的重复创建，不过这种写法却会造成内存泄漏，因为非静态内部类默认会持有外部类的引用，而该非静态内部类又创建了一个静态的实例，该实例的生命周期和应用的一样长，这就导致了该静态实例一直会持有该Activity的引用，导致Activity的内存资源不能正常回收。正确的做法为：

将该内部类设为静态内部类或将该内部类抽取出来封装成一个单例，如果需要使用Context，请按照上面推荐的使用Application 的 Context。当然，Application 的 context 不是万能的，所以也不能随便乱用，对于有些地方则必须使用 Activity 的 Context，对于Application，Service，Activity三者的Context的应用场景如下：

功能	Application	Service	Activity
Start an Activity	NO1	NO1	YES
Show a Dialog	NO	NO	YES
Layout Inflation	YES	YES	YES
Start a Service	YES	YES	YES
Bind to a Service	YES	YES	YES
Send a Broadcast	YES	YES	YES
Register BroadcastReceiver	YES	YES	YES
Load Resource Values	YES	YES	YES

其中： NO1表示 Application 和 Service 可以启动一个 Activity，不过需要创建一个新的 task 任务队列。而对于 Dialog 而言，只有在 Activity 中才能创建

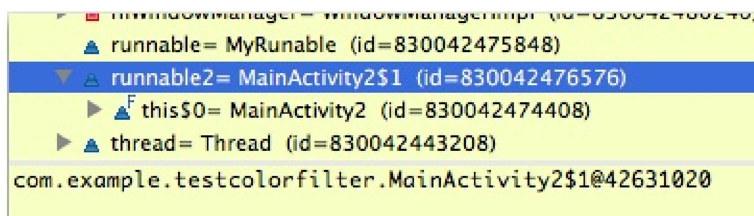
- 匿名内部类

  android开发经常会继承实现Activity/Fragment/View，此时如果你使用了匿名类，并被异步线程持有了，那要小心了，如果没有任何措施这样一定会导致泄露

```
public class MainActivity extends Activity {
  ...
  Runnable ref1 = new MyRunable();
  Runnable ref2 = new Runnable() {
      @Override
      public void run() {

      }
  };
   ...
}
```

ref1和ref2的区别是，ref2使用了匿名内部类。我们来看看运行时这两个引用的内存：



可以看到，ref1没什么特别的。 但ref2这个匿名类的实现对象里面多了一个引用：this$0这个引用指向MainActivity.this，也就是说当前的MainActivity实例会被ref2持有，如果将这个引用再传入一个异步线程，此线程和此Acitivity生命周期不一致的时候，就造成了Activity的泄露。

- Handler 造成的内存泄漏

  Handler 的使用造成的内存泄漏问题应该说是最为常见了，很多时候我们为了避免 ANR 而不在主线程进行耗时操作，在处理网络任务或者封装一些请求回调等api都借助Handler来处理，但 Handler 不是万能的，对于 Handler 的使用代码编写一不规范即有可能造成内存泄漏。另外，我们知道 Handler、Message 和 MessageQueue 都是相互关联在一起的，万一 Handler 发送的 Message 尚未被处理，则该 Message 及发送它的 Handler 对象将被线程

MessageQueue 一直持有。 由于 Handler 属于 TLS(Thread Local Storage) 变量，生命周期和 Activity 是不一致的。因此这种实现方式一般很难保证跟 View 或者 Activity 的生命周期保持一致，故很容易导致无法正确释放。

举个例子：

```
public class SampleActivity extends Activity {

  private final Handler mLeakyHandler = new Handler() {
    @Override
    public void handleMessage(Message msg) {
      // ...
    }
  }

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    // Post a message and delay its execution for 10 minutes.

    mLeakyHandler.postDelayed(new Runnable() {
      @Override
      public void run() { /* ... */ }
    }, 1000 * 60 * 10);

    // Go back to the previous Activity.
    finish();
  }
}
```

在该 SampleActivity 中声明了一个延迟10分钟执行的消息 Message，mLeakyHandler 将其 push 进了消息队列 MessageQueue 里。当该 Activity 被 finish() 掉时，延迟执行任务的 Message 还会继续存在于主线程中，它持有该 Activity 的 Handler 引用，所以此时 finish() 掉的 Activity 就不会被回收了从而造成内存泄漏（因 Handler 为非静态内部类，它会持有外部类的引用，在这里就是指 SampleActivity）。

修复方法：在 Activity 中避免使用非静态内部类，比如上面我们将 Handler 声明为静态的，则其存活期跟 Activity 的生命周期就无关了。同时通过弱引用的方式引入 Activity，避免直接将 Activity 作为 context 传进去，见下面代码：

```
public class SampleActivity extends Activity {

  /**
   * Instances of static inner classes do not hold an implicit
   * reference to their outer class.
   */
  private static class MyHandler extends Handler {
    private final WeakReference<SampleActivity> mActivity;

    public MyHandler(SampleActivity activity) {
      mActivity = new WeakReference<SampleActivity>(activity);
    }

    @Override
    public void handleMessage(Message msg) {
      SampleActivity activity = mActivity.get();
      if (activity != null) {
        // ...
      }
    }
  }

  private final MyHandler mHandler = new MyHandler(this);

  /**
   * Instances of anonymous classes do not hold an implicit
   * reference to their outer class when they are "static".
   */
  private static final Runnable sRunnable = new Runnable() {
      @Override
      public void run() { /* ... */ }
  };

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
```

```
        // Post a message and delay its execution for 10 minutes.
        mHandler.postDelayed(sRunnable, 1000 * 60 * 10);

        // Go back to the previous Activity.
        finish();
    }
}
```

综述，即推荐使用静态内部类 + WeakReference 这种方式。每次使用前注意判空。

前面提到了 WeakReference，所以这里就简单的说一下 Java 对象的几种引用类型。

Java对引用的分类有 Strong reference, SoftReference, WeakReference, PhatomReference 四种。

级别	回收时机	用途	生存时间
强	从来不会	对象的一般状态	JVM停止运行时终止
软	在内存不足时	联合ReferenceQueue构造有效期短/占内存大/生命周期长的对象的二级高速缓冲器（内存不足才清空）	内存不足时终止
弱	在垃圾回收时	联合ReferenceQueue构造有效期短/占内存大/生命周期长的对象的一级高速缓冲器（系统发生gc则清空）	gc运行后终止
虚	在垃圾回收时	联合ReferenceQueue来跟踪对象被垃圾回收器回收的活动	gc运行后终止

在Android应用的开发中，为了防止内存溢出，在处理一些占用内存大而且声明周期较长的对象时候，可以尽量应用软引用和弱引用技术。

软/弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被垃圾回收器回收，Java虚拟机就会把这个软引用加入到与之关联的引用队列中。利用这个队列可以得知被回收的软/弱引用的对象列表，从而为缓冲器清除已失效的软/弱引用。

假设我们的应用会用到大量的默认图片，比如应用中有默认的头像，默认游戏图标等等，这些图片很多地方会用到。如果每次都去读取图片，由于读取文件需要硬件操作，速度较慢，会导致性能较低。所以我们考虑将图片缓存起来，需要的时候直接从内存中读取。但是，由于图片占用内存空间比较大，缓存很多图片需要很多的内存，就可能比较容易发生OutOfMemory异常。这时，我们可以考虑使用软/弱引用技术来避免这个问题发生。以下就是高速缓冲器的雏形：

首先定义一个HashMap，保存软引用对象。

```
private Map <String, SoftReference<Bitmap>> imageCache = new Has
hMap <String, SoftReference<Bitmap>> ();
```

再来定义一个方法，保存Bitmap的软引用到HashMap。

```
public class CacheBySoftRef {
    // 首先定义一个HashMap，保存软引用对象。
    private Map<String, SoftReference<Bitmap>> imageCache = new HashMap<String,
SoftReference<Bitmap>>();
    // 再来定义一个方法，保存Bitmap的软引用到HashMap。
    public void addBitmapToCache(String path) {
        // 强引用的Bitmap对象
        Bitmap bitmap = BitmapFactory.decodeFile(path);
        // 软引用的Bitmap对象
        SoftReference<Bitmap> softBitmap = new SoftReference<Bitmap>(bitmap);
        // 添加该对象到Map中使其缓存
        imageCache.put(path, softBitmap);
    }
    // 获取的时候，可以通过SoftReference的get()方法得到Bitmap对象。
    public Bitmap getBitmapByPath(String path) {
        // 从缓存中取软引用的Bitmap对象
        SoftReference<Bitmap> softBitmap = imageCache.get(path);
        // 判断是否存在软引用
        if (softBitmap == null) {
            return null;
        }
        // 通过软引用取出Bitmap对象，如果由于内存不足Bitmap被回收，将取得空 ，如果未被回收，则可重复使
用，提高速度。
        Bitmap bitmap = softBitmap.get();
        return bitmap;
    }
}
```

使用软引用以后，在OutOfMemory异常发生之前，这些缓存的图片资源的内存空间可以被释放掉的，从而避免内存达到上限，避免Crash发生。

如果只是想避免OutOfMemory异常的发生，则可以使用软引用。如果对于应用的性能更在意，想尽快回收一些占用内存比较大的对象，则可以使用弱引用。

另外可以根据对象是否经常使用来判断选择软引用还是弱引用。如果该对象可能会经常使用的，就尽量用软引用。如果该对象不被使用的可能性更大些，就可以用弱引用。

ok，继续回到主题。前面所说的，创建一个静态Handler内部类，然后对 Handler 持有的对象使用弱引用，这样在回收时也可以回收 Handler 持有的对象，但是这样做虽然避免了 Activity 泄漏，不过 Looper 线程的消息队列中还是可能会有待处理的消息，所以我们在 Activity 的 Destroy 时或者 Stop 时应该移除消息队列 MessageQueue 中的消息。

下面几个方法都可以移除 Message：

```
public final void removeCallbacks(Runnable r);

public final void removeCallbacks(Runnable r, Object token);

public final void removeCallbacksAndMessages(Object token);

public final void removeMessages(int what);

public final void removeMessages(int what, Object object);
```

- 尽量避免使用 static 成员变量

  如果成员变量被声明为 static，那我们都知道其生命周期将与整个app进程生命周期一样。

  这会导致一系列问题，如果你的app进程设计上是长驻内存的，那即使app切到后台，这部分内存也不会被释放。按照现在手机app内存管理机制，占内存较大的后台进程将优先回收，因为如果此app做过进程互保保活，那会造成app在后台频繁重启。当手机安装了你参与开发的app以后一夜时间手机被消耗空了电量、流量，你的app不得不被用户卸载或者静默。 这里修复的方法是：

不要在类初始时初始化静态成员。可以考虑lazy初始化。 架构设计上要思考是否真的有必要这样做，尽量避免。如果架构需要这么设计，那么此对象的生命周期你有责任管理起来。

- 避免 override finalize()

  1、finalize 方法被执行的时间不确定，不能依赖与它来释放紧缺的资源。时间不确定的原因是：

  - 虚拟机调用GC的时间不确定
  - Finalize daemon线程被调度到的时间不确定

2、finalize 方法只会被执行一次，即使对象被复活，如果已经执行过了 finalize 方法，再次被 GC 时也不会再执行了，原因是：

含有 finalize 方法的 object 是在 new 的时候由虚拟机生成了一个 finalize reference 在来引用到该Object的，而在 finalize 方法执行的时候，该 object 所对应的 finalize Reference 会被释放掉，即使在这个时候把该 object 复活(即用强引用引用住该 object)，再第二次被 GC 的时候由于没有了 finalize reference 与之对应，所以 finalize 方法不会再执行。

3、含有Finalize方法的object需要至少经过两轮GC才有可能被释放。

详情见这里 深入分析过dalvik的代码

- 资源未关闭造成的内存泄漏

  对于使用了BraodcastReceiver，ContentObserver，File，游标 Cursor，Stream，Bitmap等资源的使用，应该在Activity销毁时及时关闭或者注销，否则这些资源将不会被回收，造成内存泄漏。

- 一些不良代码造成的内存压力

  有些代码并不造成内存泄露，但是它们，或是对没使用的内存没进行有效及时的释放，或是没有有效的利用已有的对象而是频繁的申请新内存。

  比如：

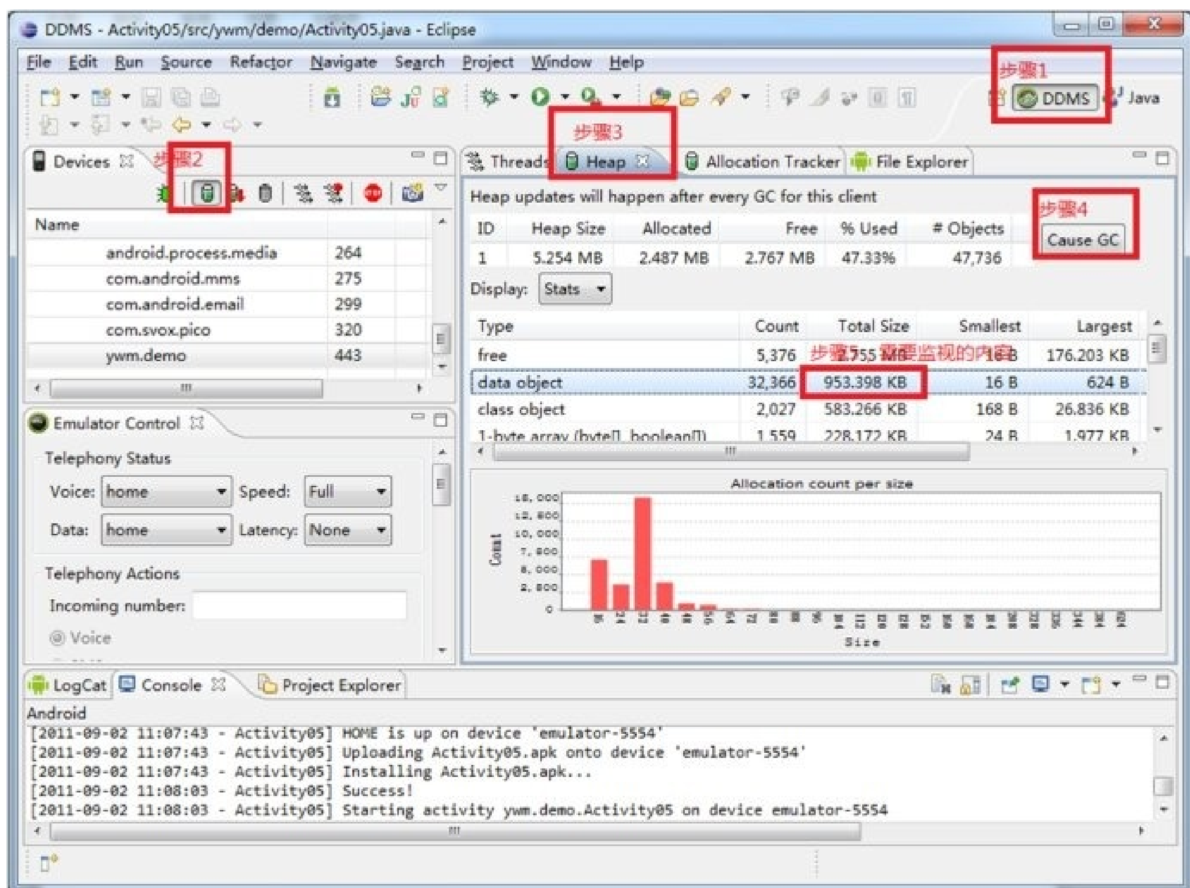
  - 构造 Adapter 时，没有使用缓存的 convertView ,每次都在创建新的 converView。这里推荐使用 ViewHolder。

工具分析

Java 内存泄漏的分析工具有很多，但众所周知的要数 MAT(Memory Analysis Tools) 和 YourKit 了。由于篇幅问题，我这里就只对 MAT 的使用做一下介绍。--> MAT 的安装

- MAT分析heap的总内存占用大小来初步判断是否存在泄露

  打开 DDMS 工具，在左边 Devices 视图页面选中“Update Heap”图标，然后在右边切换到 Heap 视图，点击 Heap 视图中的“Cause GC”按钮，到此为止需检测的进程就可以被监视。



Heap视图中部有一个Type叫做data object，即数据对象，也就是我们的程序中大量存在的类类型的对象。在data object一行中有一列是“Total Size”，其值就是当前进程中所有Java数据对象的内存总量，一般情况下，这个值的大小决定了是否会有内存泄漏。可以这样判断：

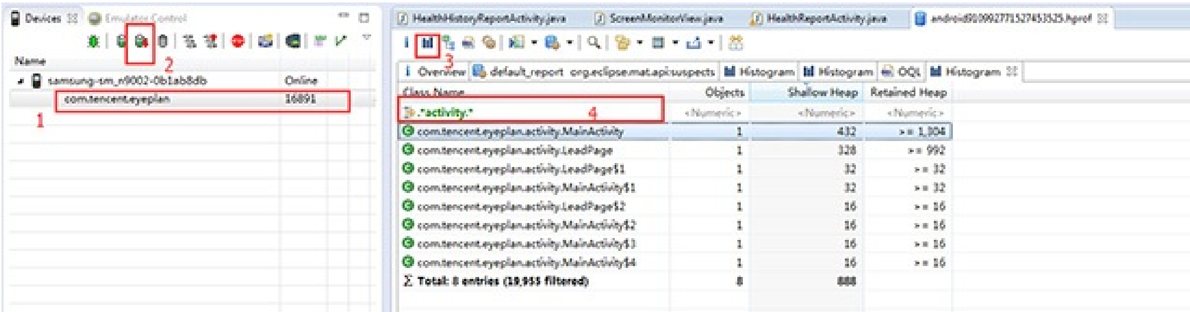
进入某应用，不断的操作该应用，同时注意观察data object的Total Size值，正常情况下Total Size值都会稳定在一个有限的范围内，也就是说由于程序中的的代码良好，没有造成对象不被垃圾回收的情况。

所以说虽然我们不断的操作会不断的生成很多对象，而在虚拟机不断的进行GC的过程中，这些对象都被回收了，内存占用量会会落到一个稳定的水平；反之如果代码中存在没有释放对象引用的情况，则data object的Total Size值在每次GC后不会有明显的回落。随着操作次数的增多Total Size的值会越来越大，直到到达一个上限后导致进程被杀掉。

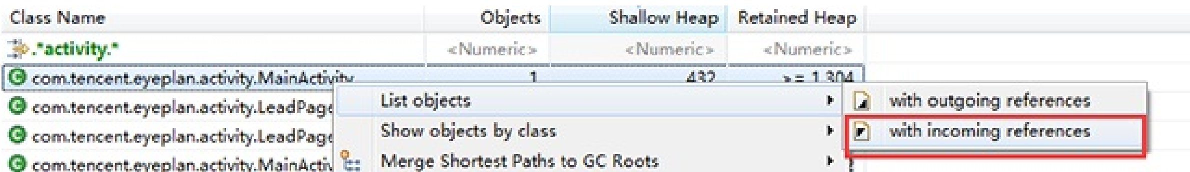
- MAT分析hprof来定位内存泄露的原因所在

这是出现内存泄露后使用MAT进行问题定位的有效手段。

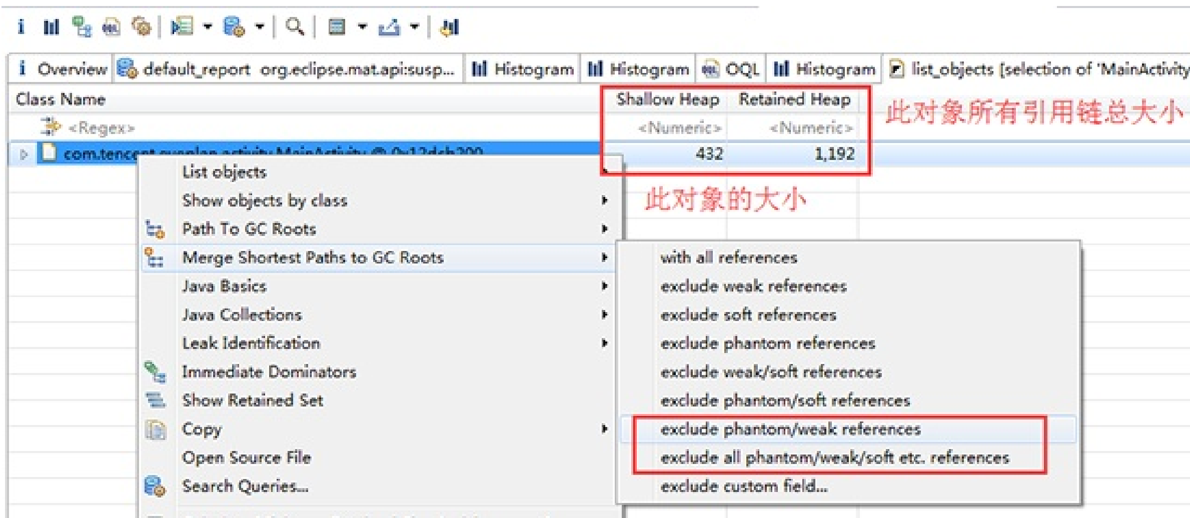
A)Dump出内存泄露当时的内存镜像hprof，分析怀疑泄露的类：



B)分析持有此类对象引用的外部对象



C)分析这些持有引用的对象的GC路径



D)逐个分析每个对象的GC路径是否正常

Class Name	Ref. Objects	Sh
<Regex>	<Numeric>	
class com.tencent.eyeplan.util.AntiRadiationUtil @ 0x12c14400 System Class	1	
mContext com.tencent.eyeplan.activity.MainActivity @ 0x12dcb200	1	

从这个路径可以看出是一个antiRadiationUtil工具类对象持有了MainActivity的引用导致MainActivity无法释放。此时就要进入代码分析此时antiRadiationUtil的引用持有是否合理（如果antiRadiationUtil持有了MainActivity的context导致节目退出后MainActivity无法销毁，那一般都属于内存泄露了）。

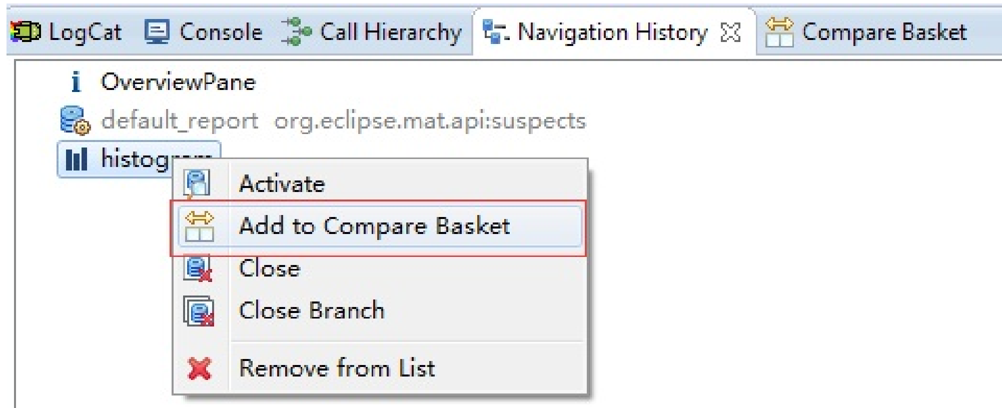
- MAT对比操作前后的hprof来定位内存泄露的根因所在

为查找内存泄漏，通常需要两个 Dump结果作对比，打开 Navigator History面板，将两个表的 Histogram结果都添加到 Compare Basket中去

A） 第一个HPROF 文件(usingFile > Open Heap Dump).

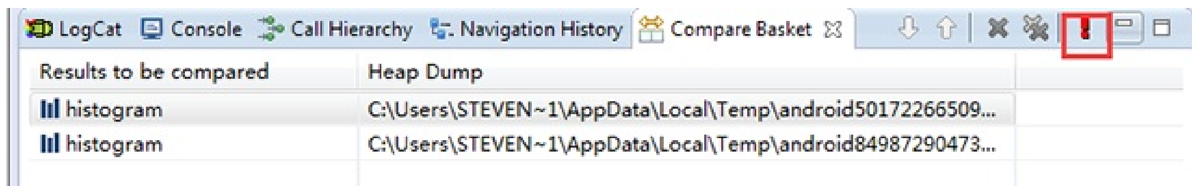
B）打开Histogram view.

C）在NavigationHistory view里 (如果看不到就从Window >show view>MAT- Navigation History), 右击histogram然后选择Add to Compare Basket .

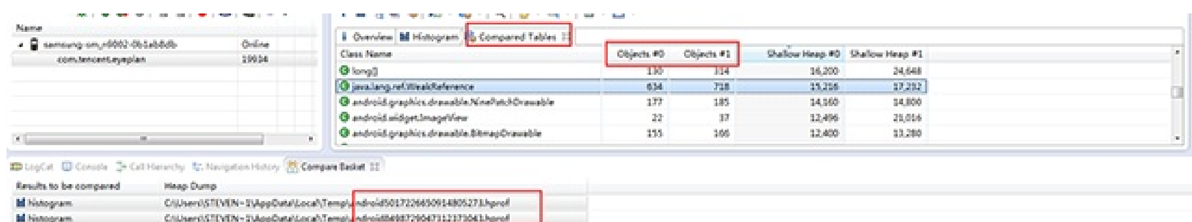


D）打开第二个HPROF 文件然后重做步骤2和3.

E）切换到Compare Basket view, 然后点击Compare the Results (视图右上角的红色"!"图标)。



F）分析对比结果



可以看出两个hprof的数据对象对比结果。

通过这种方式可以快速定位到操作前后所持有的对象增量，从而进一步定位出当前操作导致内存泄露的具体原因是泄露了什么数据对象。

注意：

如果是用 MAT Eclipse 插件获取的 Dump文件，不需要经过转换则可在MAT中打开，Adt会自动进行转换。

而手机SDk Dump 出的文件要经过转换才能被 MAT识别，Android SDK提供了这个工具 hprof-conv (位于 sdk/tools下)

首先，要通过控制台进入到你的 android sdk tools 目录下执行以下命令：

./hprof-conv xxx-a.hprof xxx-b.hprof

例如 hprof-conv input.hprof out.hprof

此时才能将out.hprof放在eclipse的MAT中打开。

Ok，下面将给大家介绍一个屌炸天的工具 -- LeakCanary 。

使用 LeakCanary 检测 Android 的内存泄漏

什么是 LeakCanary 呢？为什么选择它来检测 Android 的内存泄漏呢？

别急，让我来慢慢告诉大家！

LeakCanary 是国外一位大神 Pierre-Yves Ricau 开发的一个用于检测内存泄露的开源类库。一般情况下，在对战内存泄露中，我们都会经过以下几个关键步骤：

1、了解 OutOfMemoryError 情况。

2、重现问题。

3、在发生内存泄露的时候，把内存 Dump 出来。

4、在发生内存泄露的时候，把内存 Dump 出来。

5、计算这个对象到 GC roots 的最短强引用路径。

6、确定引用路径中的哪个引用是不该有的，然后修复问题。

很复杂对吧？

如果有一个类库能在发生 OOM 之前把这些事情全部都搞定，然后你只要修复这些问题就好了。LeakCanary 做的就是这件事情。你可以在 debug 包中轻松检测内存泄露。

一起来看这个例子（摘自 LeakCanary 中文使用说明，下面会附上所有的参考文档链接）：

```
class Cat {
}

class Box {
  Cat hiddenCat;
}
class Docker {
    // 静态变量，将不会被回收，除非加载 Docker 类的 ClassLoader 被回收。

    static Box container;
}

// ...

Box box = new Box();

// 薛定谔之猫
Cat schrodingerCat = new Cat();
box.hiddenCat = schrodingerCat;
Docker.container = box;
```

创建一个RefWatcher，监控对象引用情况。

```
// 我们期待薛定谔之猫很快就会消失（或者不消失），我们监控一下
refWatcher.watch(schrodingerCat);
```

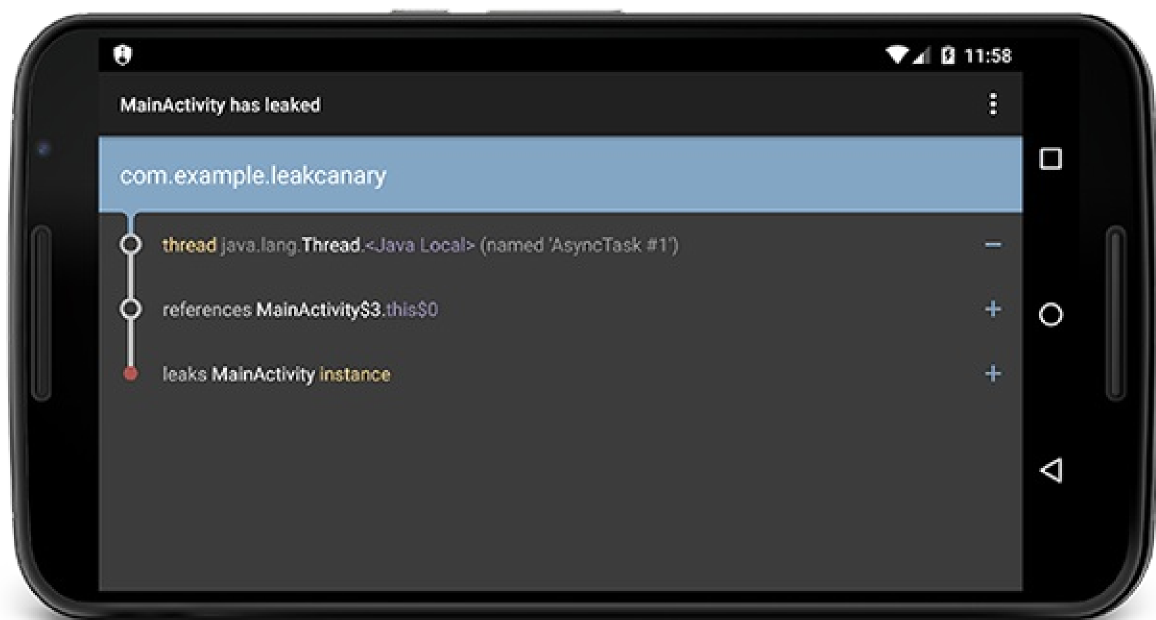
当发现有内存泄露的时候，你会看到一个很漂亮的 leak trace 报告:

- GC ROOT static Docker.container
- references Box.hiddenCat
- leaks Cat instance

我们知道，你很忙，每天都有一大堆需求。所以我们把这个事情弄得很简单，你只需要添加一行代码就行了。然后 LeakCanary 就会自动侦测 activity 的内存泄露了。

```
public class ExampleApplication extends Application {
  @Override public void onCreate() {
    super.onCreate();
    LeakCanary.install(this);
  }
}
```

然后你会在通知栏看到这样很漂亮的一个界面:



以很直白的方式将内存泄露展现在我们的面前。

Demo

一个非常简单的 LeakCanary demo: 一个非常简单的 LeakCanary demo: https://github.com/liaohuqiu/leakcanary-demo

接入

在 build.gradle 中加入引用，不同的编译使用不同的引用：

```
dependencies {
  debugCompile 'com.squareup.leakcanary:leakcanary-android:1.3'
  releaseCompile 'com.squareup.leakcanary:leakcanary-android-no
-op:1.3'
}
```

如何使用

使用 RefWatcher 监控那些本该被回收的对象。

```
RefWatcher refWatcher = {...};

// 监控
refWatcher.watch(schrodingerCat);
```

LeakCanary.install() 会返回一个预定义的 RefWatcher，同时也会启用一个 ActivityRefWatcher，用于自动监控调用 Activity.onDestroy() 之后泄露的 activity。

在Application中进行配置 ：

```
public class ExampleApplication extends Application {

  public static RefWatcher getRefWatcher(Context context) {
    ExampleApplication application = (ExampleApplication) contex
t.getApplicationContext();
    return application.refWatcher;
  }

  private RefWatcher refWatcher;

  @Override public void onCreate() {
    super.onCreate();
    refWatcher = LeakCanary.install(this);
  }
}
```

使用 RefWatcher 监控 Fragment：

```
public abstract class BaseFragment extends Fragment {

  @Override public void onDestroy() {
    super.onDestroy();
    RefWatcher refWatcher = ExampleApplication.getRefWatcher(getActivity());
    refWatcher.watch(this);
  }
}
```

使用 RefWatcher 监控 Activity：

```
public class MainActivity extends AppCompatActivity {

    ......
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
            //在自己的应用初始Activity中加入如下两行代码
        RefWatcher refWatcher = ExampleApplication.getRefWatcher(this);
        refWatcher.watch(this);

        textView = (TextView) findViewById(R.id.tv);
        textView.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                startAsyncTask();
            }
        });

    }

    private void async() {

        startAsyncTask();
    }
```

```
    private void startAsyncTask() {
        // This async task is an anonymous class and therefore h
as a hidden reference to the outer
        // class MainActivity. If the activity gets destroyed be
fore the task finishes (e.g. rotation),
        // the activity instance will leak.
        new AsyncTask<Void, Void, Void>() {
            @Override
            protected Void doInBackground(Void... params) {
                // Do some slow work in background
                SystemClock.sleep(20000);
                return null;
            }
        }.execute();
    }


}
```

工作机制

1.RefWatcher.watch() 创建一个 KeyedWeakReference 到要被监控的对象。

2.然后在后台线程检查引用是否被清除，如果没有，调用GC。

3.如果引用还是未被清除，把 heap 内存 dump 到 APP 对应的文件系统中的一个 .hprof 文件中。

4.在另外一个进程中的 HeapAnalyzerService 有一个 HeapAnalyzer 使用HAHA 解析这个文件。

5.得益于唯一的 reference key, HeapAnalyzer 找到 KeyedWeakReference，定位内存泄露。

6.HeapAnalyzer 计算 到 GC roots 的最短强引用路径，并确定是否是泄露。如果是的话，建立导致泄露的引用链。

7.引用链传递到 APP 进程中的 DisplayLeakService， 并以通知的形式展示出来。

ok,这里就不再深入了，想要了解更多就到 作者 github 主页 这去哈。

总结

- 对 Activity 等组件的引用应该控制在 Activity 的生命周期之内； 如果不能就考虑使用 getApplicationContext 或者 getApplication，以避免 Activity 被外部长生命周期的对象引用而泄露。
- 尽量不要在静态变量或者静态内部类中使用非静态外部成员变量（包括context)，即使要使用，也要考虑适时把外部成员变量置空；也可以在内部类中使用弱引用来引用外部类的变量。
- 对于生命周期比Activity长的内部类对象，并且内部类中使用了外部类的成员变量，可以这样做避免内存泄漏：
  - 将内部类改为静态内部类
  - 静态内部类中使用弱引用来引用外部类的成员变量
- Handler 的持有的引用对象最好使用弱引用，资源释放时也可以清空 Handler 里面的消息。比如在 Activity onStop 或者 onDestroy 的时候，取消掉该 Handler 对象的 Message和 Runnable.
- 在 Java 的实现过程中，也要考虑其对象释放，最好的方法是在不使用某对象时，显式地将此对象赋值为 null，比如使用完Bitmap 后先调用 recycle()，再赋为null,清空对图片等资源有直接引用或者间接引用的数组（使用 array.clear() ; array = null）等，最好遵循谁创建谁释放的原则。
- 正确关闭资源，对于使用了BraodcastReceiver，ContentObserver，File，游标 Cursor，Stream，Bitmap等资源的使用，应该在Activity销毁时及时关闭或者注销。
- 保持对对象生命周期的敏感，特别注意单例、静态对象、全局性集合等的生命周期。

一、<include/>

标签在布局优化中是使用最多的一个标签了，它就是为了解决重复定义布局的问题。标签就相当于C、C++中的include头文件一样，把一些常用的底层的API封装起来，需要的时候引入即可。在一些开源的J2EE中许多XML配置文件也都会使用标签，将多个配置文件组合成为一个更为复杂的配置文件，如最常见的S2SH。

在以前Android开发中，由于ActionBar设计上的不统一以及兼容性问题，所以很多应用都自定义了一套自己的标题栏titlebar。标题栏我们知道在应用的每个界面几乎都会用到，在这里可以作为一个很好的示例来解释标签的使用。

下面是一个自定义的titlebar文件：

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/a
ndroid"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:background="@color/titlebar_bg">

    <ImageView android:layout_width="wrap_content"
               android:layout_height="wrap_content"
               android:src="@drawable/gafricalogo" />
</FrameLayout>
```

在应用中使用titlebar布局文件，我们通过标签,布局文件如下：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@color/app_bg"
    android:gravity="center_horizontal">

    <include layout="@layout/titlebar"/>

    <TextView android:layout_width="match_parent"
              android:layout_height="wrap_content"
              android:text="@string/hello"
              android:padding="10dp" />

    ...

</LinearLayout>
```

在标签中可以覆盖导入的布局文件root布局的布局属性（如layout_*属性）。

布局示例如下：

```
<include android:id="@+id/news_title"
         android:layout_width="match_parent"
         android:layout_height="match_parent"
         layout="@layout/title"/>
```

如果想使用标签覆盖嵌入布局root布局属性，必须同时覆盖layout_height和layout_width属性，否则会直接报编译时语法错误。

> *Layout parameter layout_height ignored unless layout_width is also specified on tag*

如果标签已经定义了id，而嵌入布局文件的root布局文件也定义了id，标签的id会覆盖掉嵌入布局文件root的id，如果include标签没有定义id则会使用嵌入文件root的id。

二、 `<merge/>`

标签都是与标签组合使用的，它的作用就是可以有效减少View树的层次来优化布局。

下面通过一个简单的示例探讨一下标签的使用，下面是嵌套布局的layout_text.xml文件：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >
    <TextView
        android:id="@+id/textView"
        android:layout_width="match_parent"
        android:text="Hello World!"
        android:layout_height="match_parent" />
</LinearLayout>
```

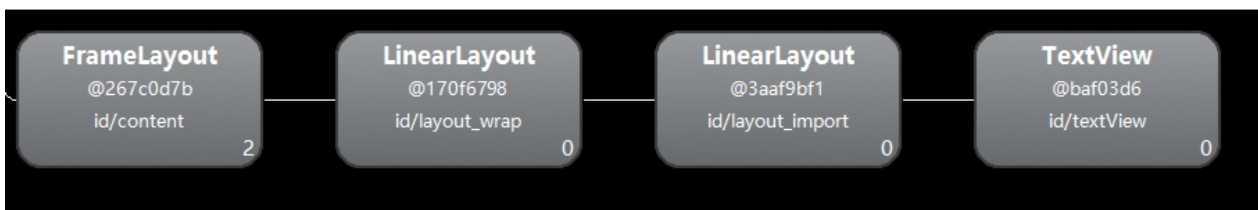
一个线性布局中嵌套一个文本视图，主布局如下：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
android"
    android:id="@+id/layout_wrap"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <include
        android:id="@+id/layout_import"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        layout="@layout/layout_text" />

</LinearLayout>
```

通过hierarchyviewer我们可以看到主布局View树的部分层级结构如下图：



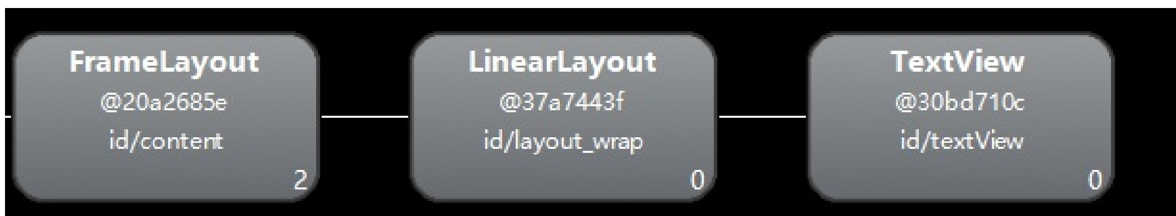
现在讲嵌套布局跟布局标签更改为，merge_text.xml布局文件如下：

```
<merge xmlns:android="http://schemas.android.com/apk/res/android"
 >

    <TextView
        android:id="@+id/textView"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:text="Hello World!"/>

</merge>
```

然后将主布局标签中的layout更改为merge_text.xml，运行后重新截图如下:



对比截图就可以发现上面的四层结构，现在已经是三层结构了。当我们使用标签的时候，系统会自动忽略merge层级，而把TextView直接放置与平级。

标签在使用的时候需要特别注意布局的类型，例如我的标签中包含的是一个LinearLayout布局视图，布局中的元素是线性排列的，如果嵌套进主布局时，include标签父布局时FrameLayout，这种方式嵌套肯定会出问题的，merge中元素会按照FrameLayout布局方式显示。所以在使用的时候，标签虽然可以减少布局层级，但是它的限制也不可小觑。

只能作为XML布局的根标签使用。当Inflate以开头的布局文件时，必须指定一个父ViewGroup，并且必须设定attachToRoot为true。

```
 View android.view.LayoutInflater.inflate(int resource, ViewGroup
root, boolean attachToRoot)
```

root不可少，attachToRoot必须为true。

三、ViewStub

在开发过程中，经常会遇到这样一种情况，有些布局很复杂但是却很少使用。例如条目详情、进度条标识或者未读消息等，这些情况如果在一开始初始化，虽然设置可见性 `View.GONE` ,但是在Inflate的时候View仍然会被Inflate，仍然会创建对象，由于这些布局又想到复杂，所以会很消耗系统资源。

ViewStub就是为了解决上面问题的，ViewStub是一个轻量级的View，它一个看不见的，不占布局位置，占用资源非常小的控件。

定义ViewStub布局文件

下面是一个ViewStub布局文件：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/layout_wrap"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <ViewStub
        android:id="@+id/stub_image"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:inflatedId="@+id/image_import"
        android:layout="@layout/layout_image" />

    <ViewStub
        android:id="@+id/stub_text"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:inflatedId="@+id/text_import"
        android:layout="@layout/layout_text" />

</LinearLayout>
```

layout_image.xml文件如下（layout_text.xml类似）：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:id="@+id/layout_image">

    <ImageView
        android:id="@+id/imageView"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />

</LinearLayout>
```

加载ViewStub布局文件

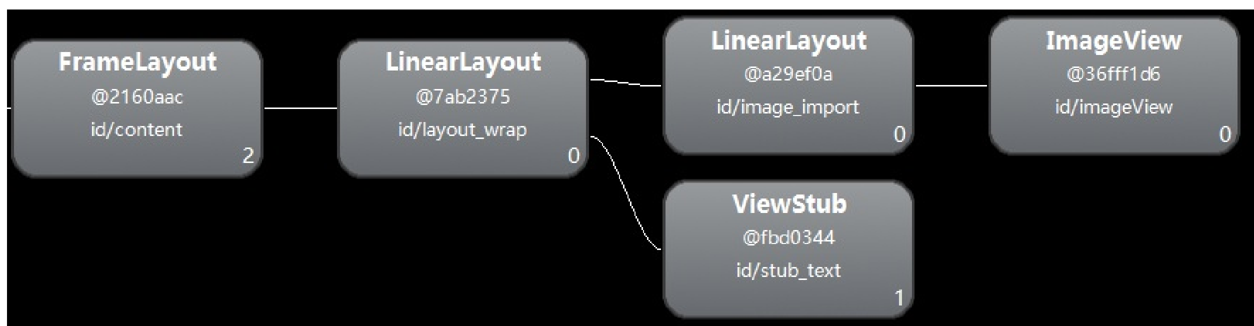
动态加载ViewStub所包含的布局文件有两种方式，方式一使用使用inflate()方法，方式二就是使用setVisibility(View.VISIBLE)。

示例java代码如下：

```
private ViewStub viewStub;

protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.layout_main2);
    viewStub = (ViewStub) findViewById(R.id.stub_image);
    //viewStub.inflate();//方式一
    viewStub.setVisibility(View.VISIBLE);//方式二
    ImageView imageView = (ImageView) findViewById(R.id.imageVie
w);
    imageView.setImageResource(R.drawable.image);
}
```

示例View层级截图如下：



ViewStub一旦visible/inflated,它自己就不在是View试图层级的一部分了。所以后面无法再使用ViewStub来控制布局，填充布局root布局如果有id，则会默认被android:inflatedId所设置的id取代，如果没有设置android:inflatedId，则会直接使用填充布局id。

由于ViewStub这种使用后即可就置空的策略，所以当需要在运行时不止一次的显示和隐藏某个布局，那么ViewStub是做不到的。这时就只能使用View的可见性来控制了。

layout_*相关属性与include标签相似，如果使用应该在ViewStub上面使用，否则使用在嵌套进来布局root上面无效。

ViewStub的另一个缺点就是目前还不支持merge标签。

四、小结

Android布局优化基本上就设计上面include、merge、ViewStub三个标签的使用。在平常开发中布局推荐使用RelativeLayout，它也可以有效减少布局层级嵌套。最后了将merge和include源码附上，ViewStub就是一个View，就不贴出来了。

Include源码

```
/**
 * Exercise <include /> tag in XML files.
 */
public class Include extends Activity {
    @Override
    protected void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        setContentView(R.layout.include_tag);
    }
}
```

Merge源码

```
/**
 * Exercise <merge /> tag in XML files.
 */
public class Merge extends Activity {
    private LinearLayout mLayout;

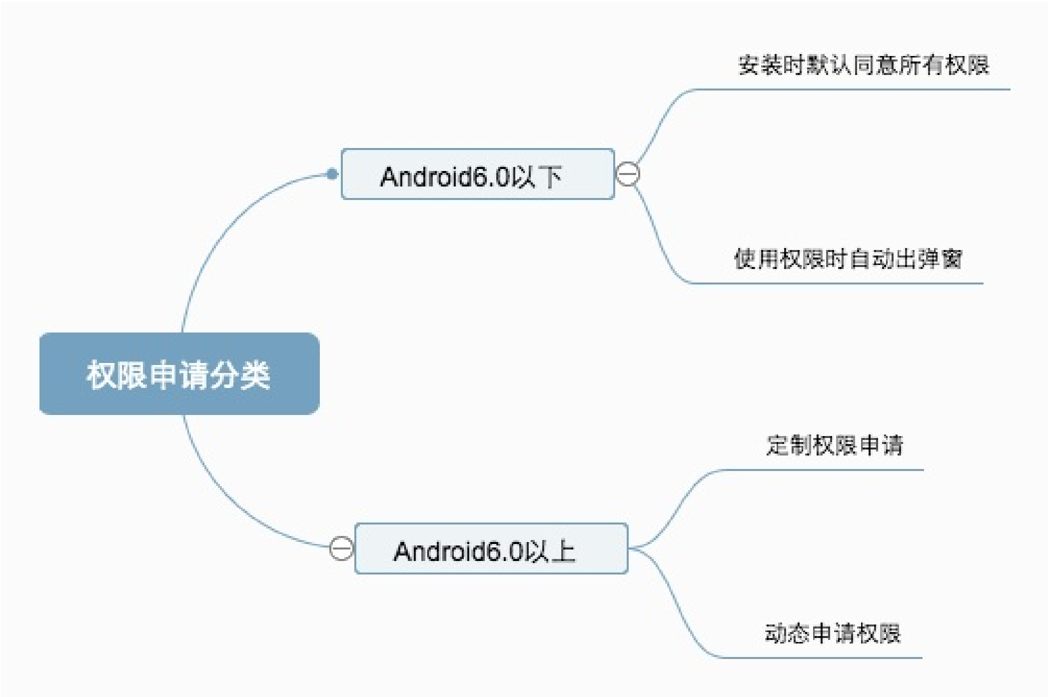
    @Override
    protected void onCreate(Bundle icicle) {
        super.onCreate(icicle);

        mLayout = new LinearLayout(this);
        mLayout.setOrientation(LinearLayout.VERTICAL);
        LayoutInflater.from(this).inflate(R.layout.merge_tag, mL
ayout);

        setContentView(mLayout);
    }

    public ViewGroup getLayout() {
        return mLayout;
    }
}
```

一、权限处理分类



由上图可以看出，主要分为四类。下表逐一介绍各类对应的一些情况。

类别	特点	适用机型
1	这是Android6.0以下系统默认的权限处理情况	原生系统手机以及一部分没有改动权限处理的系统
2	一部分定制系统会在6.0以下系统定制权限处理逻辑，在使用某个权限的时候，会弹出权限弹框，提醒用户是否同意使用该权限	vivo,oppo等手机
3	一部分定制系统会在6.0以上系统定制权限处理逻辑，不适用于动态申请，自定义实现权限处理逻辑	金立等手机，启动时出现权限请求弹窗
4	Android6.0以上系统默认的权限处理情况，动态申请权限	原生系统手机以及一部分没有改动权限处理的系统

二、动态权限申请

虽然总的来说分为四类，但是只需要处理一种情况，即动态申请权限。其他三种情况，要么默认实现，要么系统定制，无法从代码角度进行调整。那么下面先来看下那些权限需要动态申请。

(1)权限列表

CONTACTS	android.permission.WRITE_CONTACTS android.permission.GET_ACCOUNTS android.permission.READ_CONTACTS
PHONE	android.permission.READ_CALL_LOG android.permission.READ_PHONE_STATE android.permission.CALL_PHONE android.permission.WRITE_CALL_LOG android.permission.USE_SIP android.permission.PROCESS_OUTGOING_CALLS com.android.voicemail.permission.ADD_VOICEMAIL
CALENDAR	android.permission.READ_CALENDAR android.permission.WRITE_CALENDAR
CAMERA	android.permission.CAMERA
SENSORS	android.permission.BODY_SENSORS
STORAGE	android.permission.READ_EXTERNAL_STORAGE android.permission.WRITE_EXTERNAL_STORAGE
MICROPHONE	android.permission.RECORD_AUDIO
SMS	android.permission.READ_SMS android.permission.RECEIVE_WAP_PUSH android.permission.RECEIVE_MMS android.permission.RECEIVE_SMS android.permission.SEND_SMS android.permission.READ_CELL_BROADCASTS
Location	android.permission.ACCESS_COARSE_LOCATION android.permission.ACCESS_FINE_LOCATION

Android6.0以上把权限分为普通权限和危险权限，所以危险权限是需要动态申请，给予用户提示的，而危险权限就是上表展示的内容。

看到上面的 permissions，会发现一个问题，危险权限都是一组一组的。

分组对权限机制的申请是有一定影响的。例如app运行在android 6.x的机器上，对于授权机制是这样的。如果你申请某个危险的权限，假设你的app早已被用户授权了同一组的某个危险权限，那么系统会立即授权，而不需要用户去点击授权。比如你的app对READ_CONTACTS已经授权了，当你的app申请WRITE_CONTACTS时，系统会直接授权通过。

此外，对于申请时的弹窗上面的文本说明也是对整个权限组的说明，而不是单个权限。

下面介绍下Android 6.0以上 动态申请权限所设计到的一些方法。

(2)权限申请方法

在申请权限先，首先要保证在AndroidManifest中写明需要的权限。 例如：

```
<uses-permission android:name="android.permission.ACCESS_COARSE_
LOCATION"></uses-permission>
<uses-permission android:name="android.permission.ACCESS_FINE_LO
CATION"></uses-permission>
```

具体权限方法详解：

方法	参数说明	方法说明
checkSelfPermission(@NonNull Context context, @NonNull String permission)	context：上下文环境 permission:检查权限	检查是否由此权限
requestPermissions(final @NonNull String[] permissions, final @IntRange(from = 0) int requestCode)	permissions:需要申请的权限数组 requestCode:申请码，约定好，在回调时判断	用于权限的申请，系统会弹出一个申请权限的对话框
shouldShowRequestPermissionRationale(@NonNull String permission)	permission:权限名	判断是否需要向用户解释为何申请权限的原因
onRequestPermissionsResult(int requestCode, @NonNull String[] permissions, @NonNull int[] grantResults)	requestCode:请求码，即请求时规定的 permissions:权限数组 grantResults:授予权限结果 PackageManager.PERMISSION_GRANTED 同意 PackageManager.PERMISSION_DENIED 拒绝	权限请求结果的回调，判断是授予权限，还是拒绝，进行后续操作

权限申请示例

以获取定位权限为例。

1.点击按钮，检查并申请权限

```
btn.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                if (Build.VERSION.SDK_INT >23) {
                    if (ContextCompat.checkSelfPermission(MainAc
tivity.this,
                             Manifest.permission.ACCESS_COARSE_L
OCATION)
                            == PackageManager.PERMISSION_GRANTED
) {
                        //授予权限
                        getLoation();
                    }else{
                        //未获得权限
                        requestPermissions(new String[]{Manifest
.permission.ACCESS_COARSE_LOCATION}
                                ,REQUEST_CODE_LOCATION);
                    }
                }
            }
        });
```

如果有权限，执行获取位置逻辑，如果没权限，则进行请求权限。

2.权限申请结果回调

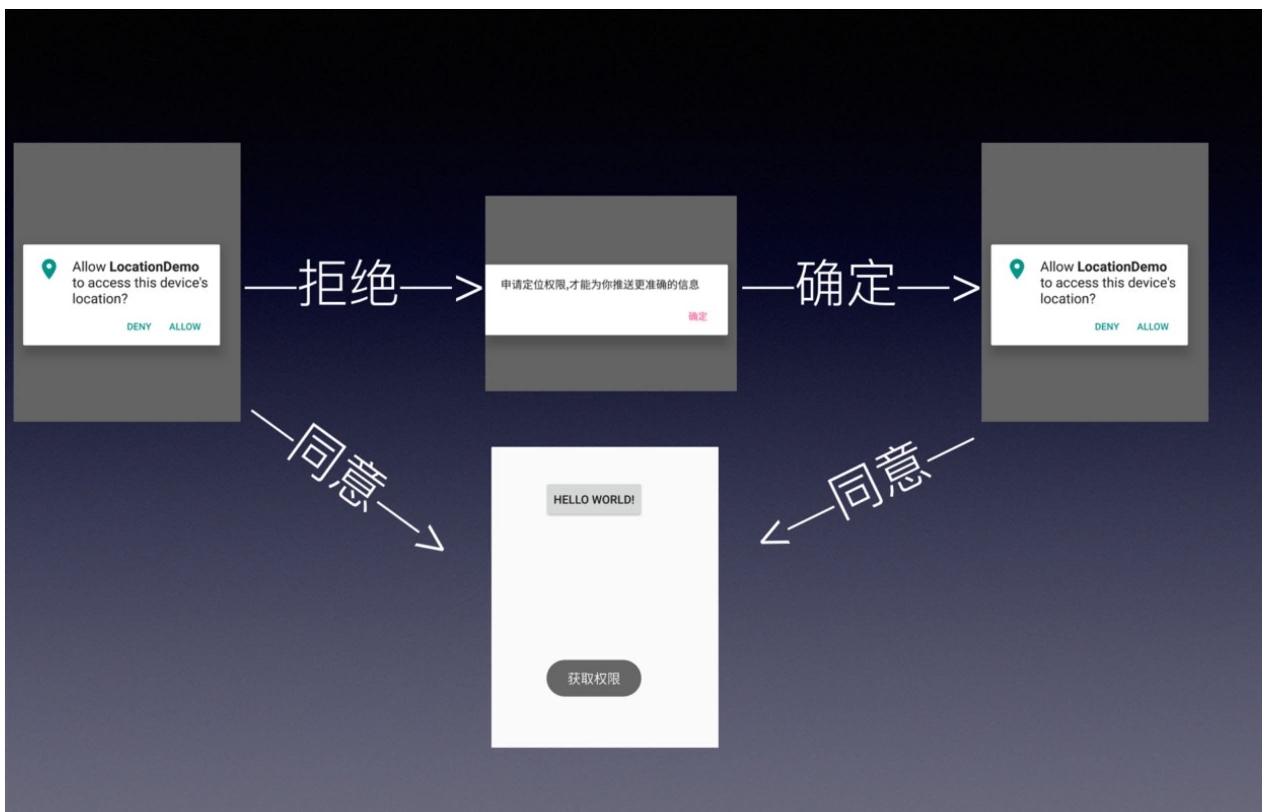
```
public void onRequestPermissionsResult(int requestCode, @NonNull
 String[] permissions, @NonNull int[] grantResults) {
        if (requestCode == REQUEST_CODE_LOCATION)
        {
            if (grantResults[0] == PackageManager.PERMISSION_GRA
NTED)
            {
                getLoation();
            } else
            {
                if (shouldShowRequestPermissionRationale( Manife
st.permission.ACCESS_COARSE_LOCATION)){
                    new AlertDialog.Builder(this)
                            .setMessage("申请定位权限,才能为你推送更
准确的信息")
                            .setPositiveButton("确定", new Dialog
Interface.OnClickListener() {
                                @Override
                                public void onClick(DialogInterf
ace dialog, int which) {
                                    //申请定位权限
                                    requestPermissions(MainActiv
ity.this,
                                            new String[]{Manifes
t.permission.ACCESS_COARSE_LOCATION}, REQUEST_CODE_LOCATION);
                                }
                            }).show();
                }
            }
            return;
        }
        super.onRequestPermissionsResult(requestCode, permission
s, grantResults);
    }
```

如果同意，执行获取位置逻辑，如果拒绝，重写shouldShowRequestPermissionRationale方法，返回true,向用户弹窗给出一个获取权限的提示，点击后再次申请权限。

```
public boolean shouldShowRequestPermissionRationale(@NonNull Str
ing permission) {
        if (permission.equals(Manifest.permission.ACCESS_COARSE_
LOCATION) ) {
            return true;
        } else {
            return super.shouldShowRequestPermissionRationale(pe
rmission);
        }
    }
```

重写shouldShowRequestPermissionRationale，在申请位置权限时，返回true，给用户解释。

以上就是动态申请权限的逻辑，大概流程如下：



注意：shouldShowRequestPermissionRationale ：默认情况下，不重写该方法，在Android原生系统中，如果第二次弹出权限申请的对话框，会出现“以后不再弹出”的提示框，如果用户勾选了，你再申请权限，则shouldShowRequestPermissionRationale返回true，意思是说要给用户一个 解释，告诉用户为什么要这个权限。

三、开源项目

- PermissionsDispatcher 使用注解的方式，动态生成类处理运行时权限.
- Grant 简化运行时权限的处理，比较灵活
- android-RuntimePermissions Google官方的例子

一、概述

最新github上开源了很多热补丁动态修复框架，大致有：

- https://github.com/dodola/HotFix
- https://github.com/jasonross/Nuwa
- https://github.com/bunnyblue/DroidFix

上述三个框架呢，根据其描述，原理都来自：安卓App热补丁动态修复技术介绍，以及Android dex分包方案，所以这俩篇务必要看。这里就不对三个框架做过多对比了，因为原理都一致，实现的代码可能差异并不是特别大。

有兴趣的直接看这篇原理文章，加上上面框架的源码基本就可以看懂了。当然了，本篇博文也会做个上述框架源码的解析，以及在整个实现过程中用到的技术的解析。

二、热修复原理

对于热修复的原理，如果你看了上面的两篇文章，相信你已经大概明白了。重点需要知道的就是，Android的ClassLoader体系，android中加载类一般使用的是 `PathClassLoader` 和 `DexClassLoader` ，首先看下这两个类的区别：

- 对于 `PathClassLoader` ，从文档上的注释来看：

  > Provides a simple {@link ClassLoader} implementation that operates on a list of files and directories in the local file system, but does not attempt to load classes from the network. Android uses this class for its system class loader and for its application class loader(s).

  可以看出，Android是使用这个类作为其系统类和应用类的加载器。并且对于这个类呢，只能去加载已经安装到Android系统中的apk文件。

- 对于 `DexClassLoader` ，依然看下注释：

  > A class loader that loads classes from {@code .jar} and {@code .apk} files containing a {@code classes.dex} entry. This can be used to execute code not installed as part of an application.

> 可以看出，该类呢，可以用来从.jar和.apk类型的文件内部加载classes.dex文件。可以用来执行非安装的程序代码。
>
> ok，如果大家对于插件化有所了解，肯定对这个类不陌生，插件化一般就是提供一个apk（插件）文件，然后在程序中load该apk，那么如何加载apk中的类呢？其实就是通过这个DexClassLoader，具体的代码我们后面有描述。

ok，到这里，大家只需要明白，Android使用PathClassLoader作为其类加载器，DexClassLoader可以从.jar和.apk类型的文件内部加载classes.dex文件就好了。

上面我们已经说了，Android使用PathClassLoader作为其类加载器，那么热修复的原理具体是？

ok，对于加载类，无非是给个classname，然后去findClass，我们看下源码就明白了。 `PathClassLoader` 和 `DexClassLoader` 都继承自 `BaseDexClassLoader` 。在BaseDexClassLoader中有如下源码：

```
#BaseDexClassLoader
@Override
protected Class<?> findClass(String name) throws ClassNotFoundEx
ception {
    Class clazz = pathList.findClass(name);

    if (clazz == null) {
        throw new ClassNotFoundException(name);
    }

    return clazz;
}

#DexPathList
public Class findClass(String name) {
    for (Element element : dexElements) {
        DexFile dex = element.dexFile;

        if (dex != null) {
            Class clazz = dex.loadClassBinaryName(name, defining
Context);
            if (clazz != null) {
                return clazz;
            }
        }
    }

    return null;
}

#DexFile
public Class loadClassBinaryName(String name, ClassLoader loader
) {
    return defineClass(name, loader, mCookie);
}
private native static Class defineClass(String name, ClassLoader
 loader, int cookie);
```

可以看出呢，BaseDexClassLoader中有个pathList对象，pathList中包含一个DexFile的集合dexElements，而对于类加载呢，就是遍历这个集合，通过DexFile去寻找。

ok，通俗点说：

> 一个ClassLoader可以包含多个dex文件，每个dex文件是一个Element，多个dex文件排列成一个有序的数组dexElements，当找类的时候，会按顺序遍历dex文件，然后从当前遍历的dex文件中找类，如果找类则返回，如果找不到从下一个dex文件继续查找。(来自：安卓App热补丁动态修复技术介绍)

那么这样的话，我们可以在这个dexElements中去做一些事情，比如，在这个数组的第一个元素放置我们的patch.jar，里面包含修复过的类，这样的话，当遍历findClass的时候，我们修复的类就会被查找到，从而替代有bug的类。

说到这，你可能已经露出笑容了，原来热修复原理这么简单。不过，还存在一个 `CLASS_ISPREVERIFIED` 的问题，对于这个问题呢，详见：安卓App热补丁动态修复技术介绍该文有图文详解。

ok，对于 `CLASS_ISPREVERIFIED` ，还是带大家理一下：

根据上面的文章，在虚拟机启动的时候，当verify选项被打开的时候，如果static方法、private方法、构造函数等，其中的直接引用（第一层关系）到的类都在同一个dex文件中，那么该类就会被打上 `CLASS_ISPREVERIFIED` 标志。

那么，我们要做的就是，阻止该类打上 `CLASS_ISPREVERIFIED` 的标志。

注意下，是阻止引用者的类，也就是说，假设你的app里面有个类叫做 `LoadBugClass` ，再其内部引用了 `BugClass` 。发布过程中发现 `BugClass` 有编写错误，那么想要发布一个新的 `BugClass` 类，那么你就要阻止 `LoadBugClass` 这个类打上 `CLASS_ISPREVERIFIED` 的标志。

也就是说，你在生成apk之前，就需要阻止相关类打上 `CLASS_ISPREVERIFIED` 的标志了。对于如何阻止，上面的文章说的很清楚，让 `LoadBugClass` 在构造方法中，去引用别的dex文件，比如：hack.dex中的某个类即可。

ok，总结下：

其实就是两件事：1、动态改变BaseDexClassLoader对象间接引用的dexElements；2、在app打包的时候，阻止相关类去打上 `CLASS_ISPREVERIFIED` 标志。

如果你没有看明白，没事，多看几遍，下面也会通过代码来说明。

三、阻止相关类打上 `CLASS_ISPREVERIFIED` 标志

ok，接下来的代码基本上会通过https://github.com/dodola/HotFix所提供的代码来讲解。

那么，这里拿具体的类来说：

大致的流程是：在dx工具执行之前，将 `LoadBugClass.class` 文件呢，进行修改，再其构造中添加 `System.out.println(dodola.hackdex.AntilazyLoad.class)` ，然后继续打包的流程。注意： `AntilazyLoad.class` 这个类是独立在hack.dex中。

ok，这里大家可能会有2个疑问：

1. 如何去修改一个类的class文件
2. 如何在dx之前去进行疑问1的操作

(1）如何去修改一个类的class文件

这里我们使用javassist来操作，很简单：

ok，首先我们新建几个类：

```
package dodola.hackdex;
public class AntilazyLoad
{

}

package dodola.hotfix;
public class BugClass
{
    public String bug()
    {
        return "bug class";
    }
}

package dodola.hotfix;
public class LoadBugClass
{
    public String getBugString()
    {
        BugClass bugClass = new BugClass();
        return bugClass.bug();
    }
}
```

注意下，这里的package，我们要做的是，上述类正常编译以后产生class文件。比如：LoadBugClass.class，我们在LoadBugClass.class的构造中去添加一行：

```
System.out.println(dodola.hackdex.AntilazyLoad.class)
```

下面看下操作类：

```
package test;

import javassist.ClassPool;
import javassist.CtClass;
import javassist.CtConstructor;

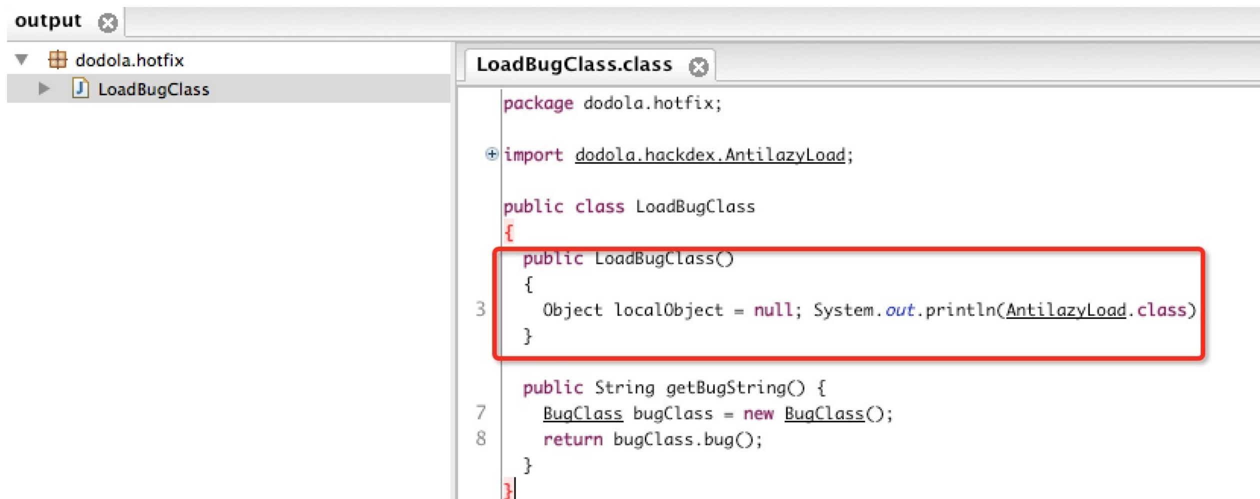
public class InjectHack
{
    public static void main(String[] args)
    {
        try
        {
            String path = "/Users/zhy/develop_work/eclipse_andro
id/imooc/JavassistTest/";
            ClassPool classes = ClassPool.getDefault();
            classes.appendClassPath(path + "bin");//项目的bin目录
即可
            CtClass c = classes.get("dodola.hotfix.LoadBugClass"
);
            CtConstructor ctConstructor = c.getConstructors()[0]
;
            ctConstructor
                    .insertAfter("System.out.println(dodola.hack
dex.AntilazyLoad.class);");
            c.writeFile(path + "/output");
        } catch (Exception e)
        {
            e.printStackTrace();
        }

    }
}
```

ok，点击run即可了，注意项目中导入javassist-*.jar的包。

首先拿到ClassPool对象，然后添加classpath，如果你有多个classpath可以多次调用。然后从classpath中找到LoadBugClass，拿到其构造方法，在其最后插入一行代码。ok，代码很好懂。

ok，我们反编译看下我们生成的class文件：



ok，关于javassist，如果有兴趣的话，大家可以参考几篇文章学习下：

- http://www.ibm.com/developerworks/cn/java/j-dyn0916/
- http://zhxing.iteye.com/blog/1703305

(2）如何在dx之前去进行(1)的操作

ok，这个就结合https://github.com/dodola/HotFix的源码来说了。

将其源码导入之后，打开app/build.gradle

```
apply plugin: 'com.android.application'

task('processWithJavassist') << {
    String classPath = file('build/intermediates/classes/debug')
//项目编译class所在目录
    dodola.patch.PatchClass.process(classPath, project(':hackdex'
).buildDir
            .absolutePath + '/intermediates/classes/debug')//第二
个参数是hackdex的class所在目录

}
android {
    applicationVariants.all { variant ->
        variant.dex.dependsOn << processWithJavassist //在执行dx
命令之前将代码打入到class中
    }
}
```

你会发现，在执行dx之前，会先执行processWithJavassist这个任务。这个任务的作用呢，就和我们上面的代码一致了。而且源码也给出了，大家自己看下。

ok，到这呢，你就可以点击run了。ok，有兴趣的话，你可以反编译去看看 `dodola.hotfix.LoadBugClass` 这个类的构造方法中是否已经添加了改行代码。

关于反编译的用法，工具等，参考：http://blog.csdn.net/lmj623565791/article/details/23564065

ok，到此我们已经能够正常的安装apk并且运行了。但是目前还未涉及到打补丁的相关代码。

四、动态改变BaseDexClassLoader对象间接引用的dexElements

ok，这里就比较简单了，动态改变一个对象的某个引用我们反射就可以完成了。

不过这里需要注意的是，还记得我们之前说的，寻找class是遍历dexElements；然后我们的 `AntilazyLoad.class` 实际上并不包含在apk的classes.dex中，并且根据上面描述的需要，我们需要将 `AntilazyLoad.class` 这个类打成独立的hack_dex.jar，注意不是普通的jar，必须经过dx工具进行转化。

具体做法:

```
jar cvf hack.jar dodola/hackdex/*
dx  --dex --output hack_dex.jar hack.jar 1212
```

如果，你没有办法把那一个class文件搞成jar，去百度一下...

ok，现在有了hack_dex.jar，这个是干嘛的呢？

应该还记得，我们的app中部门类引用了 `AntilazyLoad.class` ，那么我们必须在应用启动的时候，将这个hack_dex.jar插入到dexElements，否则肯定会出事故的。

那么，Application的onCreate方法里面就很适合做这件事情，我们把hack_dex.jar放到assets目录。

下面看hotfix的源码：

```
/*
 * Copyright (C) 2015 Baidu, Inc. All Rights Reserved.
 */
package dodola.hotfix;

import android.app.Application;
import android.content.Context;

import java.io.File;

import dodola.hotfixlib.HotFix;

/**
 * Created by sunpengfei on 15/11/4.
 */
public class HotfixApplication extends Application
{

    @Override
    public void onCreate()
    {
        super.onCreate();
        File dexPath = new File(getDir("dex", Context.MODE_PRIVA
TE), "hackdex_dex.jar");
        Utils.prepareDex(this.getApplicationContext(), dexPath,
"hackdex_dex.jar");
        HotFix.patch(this, dexPath.getAbsolutePath(), "dodola.ha
ckdex.AntilazyLoad");
        try
        {
            this.getClassLoader().loadClass("dodola.hackdex.Anti
lazyLoad");
        } catch (ClassNotFoundException e)
        {
            e.printStackTrace();
        }

    }
}
```

ok，在app的私有目录创建一个文件，然后调用Utils.prepareDex将assets中的hackdex_dex.jar写入该文件。 接下来HotFix.patch就是去反射去修改dexElements了。我们深入看下源码：

```
/*
 * Copyright (C) 2015 Baidu, Inc. All Rights Reserved.
 */
package dodola.hotfix;

/**
 * Created by sunpengfei on 15/11/4.
 */
public class Utils {
    private static final int BUF_SIZE = 2048;

    public static boolean prepareDex(Context context, File dexIn
ternalStoragePath, String dex_file) {
        BufferedInputStream bis = null;
        OutputStream dexWriter = null;
        bis = new BufferedInputStream(context.getAssets().open(d
ex_file));
        dexWriter = new BufferedOutputStream(new FileOutputStrea
m(dexInternalStoragePath));
        byte[] buf = new byte[BUF_SIZE];
        int len;
        while ((len = bis.read(buf, 0, BUF_SIZE)) > 0) {
            dexWriter.write(buf, 0, len);
        }
        dexWriter.close();
        bis.close();
        return true;

}
```

ok，其实就是文件的一个读写，将assets目录的文件，写到app的私有目录中的文件。

下面主要看patch方法

```
/*
 * Copyright (C) 2015 Baidu, Inc. All Rights Reserved.
 */
package dodola.hotfixlib;

import android.annotation.TargetApi;
import android.content.Context;

import java.io.File;
import java.lang.reflect.Array;
import java.lang.reflect.Field;
import java.lang.reflect.InvocationTargetException;

import dalvik.system.DexClassLoader;
import dalvik.system.PathClassLoader;

/* compiled from: ProGuard */
public final class HotFix
{
    public static void patch(Context context, String patchDexFil
e, String patchClassName)
    {
        if (patchDexFile != null && new File(patchDexFile).exist
s())
        {
            try
            {
                if (hasLexClassLoader())
                {
                    injectInAliyunOs(context, patchDexFile, patc
hClassName);
                } else if (hasDexClassLoader())
                {
                    injectAboveEqualApiLevel14(context, patchDex
File, patchClassName);
                } else
                {

                    injectBelowApiLevel14(context, patchDexFile,
```

```
patchClassName);

                }
            } catch (Throwable th)
            {
            }
        }
    }
}
```

这里很据系统中ClassLoader的类型做了下判断，原理都是反射，我们看其中一个分支 `hasDexClassLoader()`；

```
private static boolean hasDexClassLoader()
{
    try
    {
        Class.forName("dalvik.system.BaseDexClassLoader");
        return true;
    } catch (ClassNotFoundException e)
    {
        return false;
    }
}


 private static void injectAboveEqualApiLevel14(Context context,
 String str, String str2)
            throws ClassNotFoundException, NoSuchFieldException,
 IllegalAccessException
{
    PathClassLoader pathClassLoader = (PathClassLoader) context.
getClassLoader();
    Object a = combineArray(getDexElements(getPathList(pathClass
Loader)),
            getDexElements(getPathList(
                    new DexClassLoader(str, context.getDir("dex"
, 0).getAbsolutePath(), str, context.getClassLoader()))));
    Object a2 = getPathList(pathClassLoader);
    setField(a2, a2.getClass(), "dexElements", a);
    pathClassLoader.loadClass(str2);
}
```

首先查找类 `dalvik.system.BaseDexClassLoader` ，如果找到则进入if体。

在injectAboveEqualApiLevel14中，根据context拿到PathClassLoader，然后通过getPathList(pathClassLoader)，拿到PathClassLoader中的pathList对象，在调用getDexElements通过pathList取到dexElements对象。

ok，那么我们的hack_dex.jar如何转化为dexElements对象呢？

通过源码可以看出，首先初始化了一个DexClassLoader对象，前面我们说过DexClassLoader的父类也是BaseDexClassLoader，那么我们可以通过和PathClassLoader同样的方式取得dexElements。

ok，到这里，我们取得了，系统中PathClassLoader对象的间接引用dexElements，以及我们的hack_dex.jar中的dexElements，接下来就是合并这两个数组了。

可以看到上面的代码使用的是combineArray方法。

合并完成后，将新的数组通过反射的方式设置给pathList.

接下来看一下反射的细节：

```
private static Object getPathList(Object obj) throws ClassNotFoundException, NoSuchFieldException,
            IllegalAccessException
{
    return getField(obj, Class.forName("dalvik.system.BaseDexClassLoader"), "pathList");
}

private static Object getDexElements(Object obj) throws NoSuchFieldException, IllegalAccessException
{
    return getField(obj, obj.getClass(), "dexElements");
}

private static Object getField(Object obj, Class cls, String str)

            throws NoSuchFieldException, IllegalAccessException
{
    Field declaredField = cls.getDeclaredField(str);
    declaredField.setAccessible(true);
    return declaredField.get(obj);
}
```

其实都是取成员变量的过程，应该很容易懂~~

```
private static Object combineArray(Object obj, Object obj2)
{
    Class componentType = obj2.getClass().getComponentType();
    int length = Array.getLength(obj2);
    int length2 = Array.getLength(obj) + length;
    Object newInstance = Array.newInstance(componentType, length
2);
    for (int i = 0; i < length2; i++)
    {
        if (i < length)
        {
            Array.set(newInstance, i, Array.get(obj2, i));
        } else
        {
            Array.set(newInstance, i, Array.get(obj, i - length)
);
        }
    }
    return newInstance;
}
```

ok，这里的两个数组合并，只需要注意一件事，将hack_dex.jar里面的dexElements放到新数组前面即可。

到此，我们就完成了在应用启动的时候，动态的将hack_dex.jar中包含的DexFile注入到ClassLoader的dexElements中。这样就不会查找不到AntilazyLoad这个类了。

ok，那么到此呢，还是没有看到我们如何打补丁，哈，其实呢，已经说过了，打补丁的过程和我们注入hack_dex.jar是一致的。

你现在运行HotFix的app项目，点击menu里面的测试：

会弹出：`调用测试方法：bug class`

接下来就看如何完成热修复。

五、完成热修复

ok，那么我们假设BugClass这个类有错误，需要修复：

```
package dodola.hotfix;

public class BugClass
{
    public String bug()
    {
        return "fixed class";
    }
}
```

可以看到字符串变化了：bug class -> fixed class .

然后，编译，将这个类的class->jar->dex。步骤和上面是一致的。

```
jar cvf path.jar dodola/hotfix/BugClass.class
dx  --dex --output path_dex.jar path.jar 1212
```

拿到path_dex.jar文件。

正常情况下，这个玩意应该是下载得到的，当然我们介绍原理，你可以直接将其放置到sdcard上。

然后在Application的onCreate中进行读取，我们这里为了方便也放置到assets目录，然后在Application的onCreate中添加代码：

```
public class HotfixApplication extends Application
{

    @Override
    public void onCreate()
    {
        super.onCreate();
        File dexPath = new File(getDir("dex", Context.MODE_PRIVA
TE), "hackdex_dex.jar");
        Utils.prepareDex(this.getApplicationContext(), dexPath,
"hack_dex.jar");
        HotFix.patch(this, dexPath.getAbsolutePath(), "dodola.ha
ckdex.AntilazyLoad");
        try
        {
            this.getClassLoader().loadClass("dodola.hackdex.Anti
lazyLoad");
        } catch (ClassNotFoundException e)
        {
            e.printStackTrace();
        }

        dexPath = new File(getDir("dex", Context.MODE_PRIVATE),
"path_dex.jar");
        Utils.prepareDex(this.getApplicationContext(), dexPath,
"path_dex.jar");
        HotFix.patch(this, dexPath.getAbsolutePath(), "dodola.ho
tfix.BugClass");

    }
}
```

其实就是添加了后面的3行，这里需要说明一下，第一行依旧是复制到私有目录，如果你是sdcard上，那么操作基本是一致的，这里就别问：如果在sdcard或者网络上怎么处理~

ok，那么再次运行我们的app。

HotFix



Hello World!

测试调用方法:fixed class,zhy



ok，最后说一下，说项目中有一个打补丁的按钮，在menu下，那么你也可以不在Application里面添加我们最后的3行。

你运行app后，先点击`打补丁`，然后点击`测试`也可以发现成功修复了。

如果先点击`测试`，再点击`打补丁`，再`测试`是不会变化的，因为类一旦加载以后，不会重新再去重新加载了。

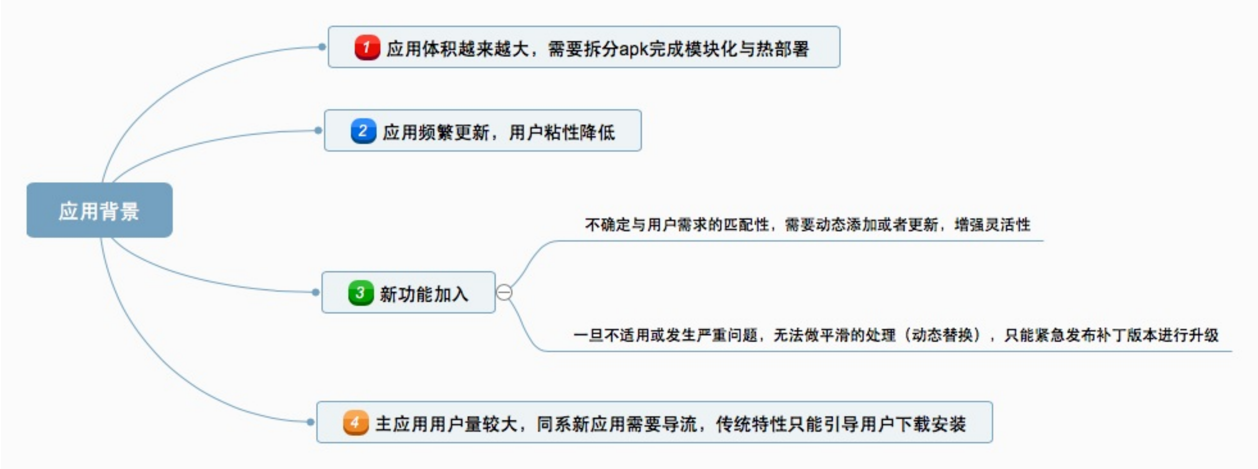
ok，到此，我们的热修复的原理，已经解决方案，我相信已经很详细的介绍完成了，如果你有足够的耐心一定可以实现。中间制作补丁等操作，我们的操作比较麻烦，自动化的话，可以参考https://github.com/jasonross/Nuwa。

一、Android插件化介绍

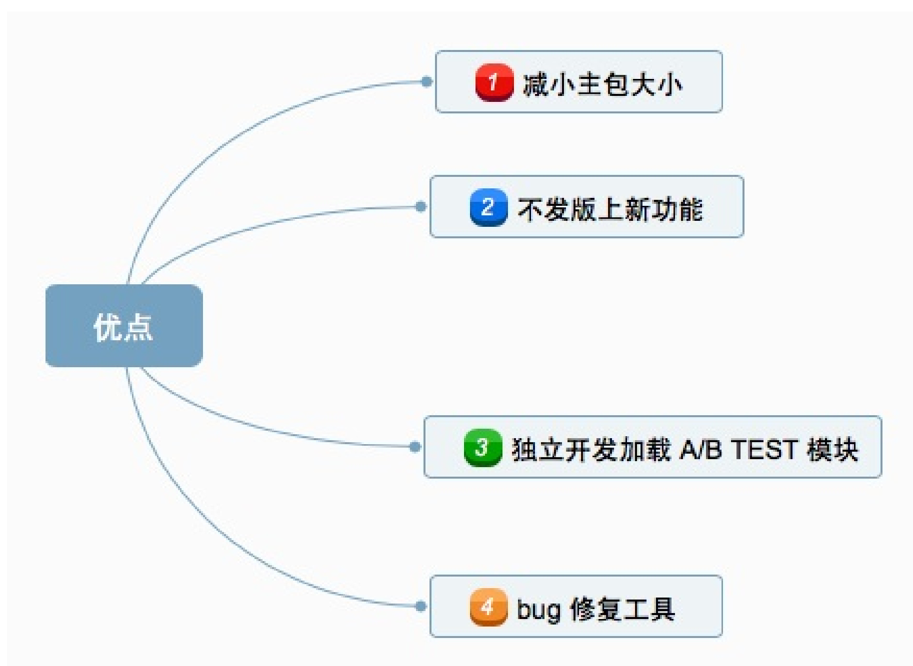
1.含义

所谓插件化，就是让我们的应用不必再像原来一样把所有的内容都放在一个**apk**中，可以把一些功能和逻辑单独抽出来放在插件apk中，然后主apk做到［按需调用］，这样的好处是一来可以减少主apk的体积，让应用更轻便，二来可以做到热插拔，更加动态化。

2.背景



3.优点



二、Android插件化基础

1.插件化的特点

1) 应用在运行的时候通过加载一些本地不存在的可执行文件实现一些特定的功能;

2) 这些可执行文件是可以替换的;

3) 更换静态资源（比如换启动图、换主题、或者用服务器参数开关控制广告的隐藏现实等）不属于动态加载;

4) Android中动态加载的核心思想是动态调用外部的 dex文件，极端的情况下，Android APK自身带有的Dex文件只是一个程序的入口（或者说空壳），所有的功能都通过从服务器下载最新的Dex文件完成;

2.需要解决的问题

代码加载

类的加载可以使用Java的ClassLoader机制，但是对于Android来说，并不是说类加载进来就可以用了，很多组件都是有“生命”的；因此对于这些有血有肉的类，必须给它们注入活力，也就是所谓的组件生命周期管理；

资源加载

资源加载方案大家使用的原理都差不多，都是用AssetManager的隐藏方法addAssetPath。

3.插件化必备基础

①ClassLoader类加载器

要想实现加载外部dex文件（即插件）来实现热部署，那么必然要把其中的class文件加载到内存中。

其中涉及到两种ClassLoader:DexClassLoader和PathClassLoader。而DexClassLoader可以加载外部的jar,dex等文件，正是我们需要的。

关于ClassLoader详解，见ClassLoader完全解析。

②Java反射

因为插件apk与宿主apk不在一个apk内，那么一些类的访问必然要通过反射进行获取。所以了解反射对插件化的学习是必须的。

关于Java反射，见Java反射详解。

③插件资源访问

res里的每一个资源都会在R.java里生成一个对应的Integer类型的id，APP启动时会先把R.java注册到当前的上下文环境，我们在代码里以R文件的方式使用资源时正是通过使用这些id访问res资源，然而插件的R.java并没有注册到当前的上下文环境，所以插件的res资源也就无法通过id使用了。

查看源码，通过“addAssetPath”方法重新生成一个新的Resource对象来保存插件中的资源，避免冲突。

关于插件资源访问，见使用插件中的R资源。

④代理模式

插件化实现的过程主要靠欺上瞒下，坑蒙拐骗来实现。想想虽然加载进来了Activity等组件，但也仅仅是最为一个对象而存在，并没有在AndroidManifest中注册，没有生命周期的回调，并不能实现我们想要的效果。因此无论是dynamic_load_apk通过

代理activity来操控插件activity的方式，还是DroidPlugin通过hook activity启动过程来启动插件activity的方式，都是对代理模式的应用。

关于代理模式，见静态代理与动态代理。

至此，通过ClassLoader加载，然后通过代理模式让Activity等组件具有生命周期实现真正的功能，并且解决了资源访问问题。可能插件化已经可以简单的实现一些初步的功能，然而插件化绝不止于此。更多的内容仍需要进一步探索，不过以上知识是基础中的基础，必备之必备。

三、Android插件化开源项目

名称	时间	作者	说明
AndroidDynamicLoader	2012.7	屠毅敏	
23Code	2013		一个可以下载很多开源项目的apk，应用市场上可以下载得到
Altas	2014年初	伯奎	
Dynamic-load-apk	2014年底	任玉刚	
OpenAltas / ACDD	2015		
DroidPlugin	2015.8	张勇	
VirtualApp	2016	罗迪	
DynamicAPK		携程	
AndFix	2015.9	黎三平	
QQ空间超级补丁			
微信Tinker			
Nuwa	2015.11	贾吉鑫	

介绍一下其中比较重要的两个，实现思想不同，也是入门插件化可以学习的两个。

Dynamic-load-apk

Dynamic-Load-Apk简称DL，这个开源框架作者是任玉刚，他的实现方式是，在宿主中埋一个代理Activity，更改ClassLoader后找到加载插件中的Activity，使用宿主中的Activity作为代理，回调给插件中Activity所以对应的生命周期。这个思路与

AndroidDynamicLoader有点像，都是做一个代理，只不过Dynamic-load-apk加载的插件中的Activity。 项目地址：https://github.com/singwhatiwanna/dynamic-load-apk

DroidPlugin

DroidPlugin是张勇实现的一套插件化方案，它的原理是Hook客户端一侧的系统Api。 项目地址：https://github.com/DroidPluginTeam/DroidPlugin

既然着重介绍了两个项目，必然要学起来，怎么学习呢？

好在已经有前人把自己的学习经验分享出来，那么我们只需要结合源码进行学习即可。

四、Dynamic-load-apk详解

Android插件化学习之路（一）之动态加载综述

Android插件化学习之路（二）之ClassLoader完全解析

Android插件化学习之路（三）之调用外部.dex文件中的代码

Android插件化学习之路（四）之使用插件中的R资源

Android插件化学习之路（五）之代理Activity

Android插件化学习之路（六）之动态创建Activity

Android插件化学习之路（七）之DL插件开发该注意的坑

Android插件化学习之路（八）之DynamicLoadApk 源码解析（上）

Android插件化学习之路（九）之DynamicLoadApk 源码解析（下）

五、DroidPlugin详解

Hook机制之动态代理

Hook机制之Binder Hook

Hook机制之AMS&PMS

Activity生命周期管理

插件加载机制

广播的管理

Service的插件化

ContentProvider的插件化

一、概述

之前一直没有写过插件化相关的博客，刚好最近滴滴和360分别开源了自家的插件化方案，赶紧学习下，写两篇博客，第一篇是滴滴的方案：

- https://github.com/didi/VirtualAPK

那么其中的难点很明显是对四大组件支持，因为大家都清楚，四大组件都是需要在AndroidManifest中注册的，而插件apk中的组件是不可能预先知晓名字，提前注册中宿主apk中的，所以现在基本都采用一些hack方案类解决，VirtualAPK大体方案如下：

- Activity：在宿主apk中提前占几个坑，然后通过“欺上瞒下”（这个词好像是360之前的ppt中提到）的方式，启动插件apk的Activity；因为要支持不同的launchMode以及一些特殊的属性，需要占多个坑。
- Service：通过代理Service的方式去分发；主进程和其他进程，VirtualAPK使用了两个代理Service。
- BroadcastReceiver：静态转动态
- ContentProvider：通过一个代理Provider进行分发。

这些占坑的数量并不是固定的，比如Activity想支持某个属性，该属性不能动态设置，只能在Manifest中设置，那就需要去占坑支持。所以占坑数量这些，可以根据自己的需求进行调整。

下面就逐一去分析代码啦~

> 注：本篇博客涉及到的framework逻辑，为API 22. 分期版本为com.didi.virtualapk:core:0.9.0

二、Activity的支持

这里就不按照某个流程一行行代码往下读了，针对性的讲一些关键流程，可能更好阅读一些。

首先看一段启动插件Activity的代码：

```
final String pkg = "com.didi.virtualapk.demo";
if (PluginManager.getInstance(this).getLoadedPlugin(pkg) == null
) {
    Toast.makeText(this, "plugin [com.didi.virtualapk.demo] not
loaded", Toast.LENGTH_SHORT).show();
    return;
}

// test Activity and Service
Intent intent = new Intent();
intent.setClassName(pkg, "com.didi.virtualapk.demo.aidl.BookMana
gerActivity");
startActivity(intent);
```

可以看到优先根据包名判断该插件是否已经加载，所以在插件使用前其实还需要调用

```
pluginManager.loadPlugin(apk);
```

加载插件。

这里就不赘述源码了，大致为调用 `PackageParser.parsePackage` 解析apk，获得该apk对应的PackageInfo，资源相关（AssetManager，Resources），DexClassLoader（加载类），四大组件相关集合（mActivityInfos，mServiceInfos，mReceiverInfos，mProviderInfos），针对Plugin的PluginContext等一堆信息，封装为LoadedPlugin对象。

> 详细可以参考 `com.didi.virtualapk.internal.LoadedPlugin` 类。

ok，如果该插件以及加载过，则直接通过startActivity去启动插件中目标Activity。

（1）替换Activity

这里大家肯定会有疑惑，该Activity必然没有在Manifest中注册，这么启动不会报错吗？

正常肯定会报错呀，所以我们看看它是怎么做的吧。

跟进startActivity的调用流程，会发现其最终会进入Instrumentation的execStartActivity方法，然后再通过ActivityManagerProxy与AMS进行交互。

而Activity是否存在的校验是发生在AMS端，所以我们在于AMS交互前，提前将Activity的ComponentName进行替换为占坑的名字不就好了么？

这里可以选择hook Instrumentation，或者ActivityManagerProxy都可以达到目标，VirtualAPK选择了hook Instrumentation.

打开 `PluginManager` 可以看到如下方法：

```
private void hookInstrumentationAndHandler() {
    try {
        Instrumentation baseInstrumentation = ReflectUtil.getInstrumentation(this.mContext);
        if (baseInstrumentation.getClass().getName().contains("lbe")) {
            // reject executing in paralell space, for example, lbe.
            System.exit(0);
        }

        final VAInstrumentation instrumentation = new VAInstrumentation(this, baseInstrumentation);
        Object activityThread = ReflectUtil.getActivityThread(this.mContext);
        ReflectUtil.setInstrumentation(activityThread, instrumentation);
        ReflectUtil.setHandlerCallback(this.mContext, instrumentation);
        this.mInstrumentation = instrumentation;
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

可以看到首先通过反射拿到了原本的 `Instrumentation` 对象，拿的过程是首先拿到ActivityThread，由于ActivityThread可以通过静态变量 `sCurrentActivityThread` 或者静态方法 `currentActivityThread()` 获取，

所以拿到其对象相当轻松。拿到ActivityThread对象后，调用其 `getInstrumentation()` 方法，即可获取当前的Instrumentation对象。

然后自己创建了一个VAInstrumentation对象，接下来就直接反射将VAInstrumentation对象设置给ActivityThread对象即可。

这样就完成了hook Instrumentation,之后调用Instrumentation的任何方法，都可以在VAInstrumentation进行拦截并做一些修改。

这里还hook了ActivityThread的mH类的Callback，暂不赘述。

刚才说了，可以通过Instrumentation的execStartActivity方法进行偷梁换柱，所以我们直接看对应的方法：

```
public ActivityResult execStartActivity(
        Context who, IBinder contextThread, IBinder token, Activ
ity target,
        Intent intent, int requestCode, Bundle options) {
    mPluginManager.getComponentsHandler().transformIntentToExpli
citAsNeeded(intent);
    // null component is an implicitly intent
    if (intent.getComponent() != null) {
        Log.i(TAG, String.format("execStartActivity[%s : %s]", i
ntent.getComponent().getPackageName(),
                intent.getComponent().getClassName()));
        // resolve intent with Stub Activity if needed
        this.mPluginManager.getComponentsHandler().markIntentIfN
eeded(intent);
    }

    ActivityResult result = realExecStartActivity(who, contextTh
read, token, target,
                intent, requestCode, options);

    return result;

}
```

首先调用transformIntentToExplicitAsNeeded，这个主要是当component为null时，根据启动Activity时，配置的action，data,category等去已加载的plugin中匹配到确定的Activity的。

本例我们的写法ComponentName肯定不为null，所以直接看 `markIntentIfNeeded()` 方法：

```
public void markIntentIfNeeded(Intent intent) {
    if (intent.getComponent() == null) {
        return;
    }

    String targetPackageName = intent.getComponent().getPackageN
ame();
    String targetClassName = intent.getComponent().getClassName(
);
    // search map and return specific launchmode stub activity
    if (!targetPackageName.equals(mContext.getPackageName())
            && mPluginManager.getLoadedPlugin(targetPackageName)
 != null) {
        intent.putExtra(Constants.KEY_IS_PLUGIN, true);
        intent.putExtra(Constants.KEY_TARGET_PACKAGE, targetPack
ageName);
        intent.putExtra(Constants.KEY_TARGET_ACTIVITY, targetCla
ssName);
        dispatchStubActivity(intent);
    }
}
```

在该方法中判断如果启动的是插件中类，则将启动的包名和Activity类名存到了intent中，可以看到这里存储明显是为了后面恢复用的。

然后调用了 `dispatchStubActivity(intent)`

```
private void dispatchStubActivity(Intent intent) {
    ComponentName component = intent.getComponent();
    String targetClassName = intent.getComponent().getClassName(
);
    LoadedPlugin loadedPlugin = mPluginManager.getLoadedPlugin(i
ntent);
    ActivityInfo info = loadedPlugin.getActivityInfo(component);
    if (info == null) {
        throw new RuntimeException("can not find " + component);
    }
    int launchMode = info.launchMode;
    Resources.Theme themeObj = loadedPlugin.getResources().newTh
eme();
    themeObj.applyStyle(info.theme, true);
    String stubActivity = mStubActivityInfo.getStubActivity(targ
etClassName, launchMode, themeObj);
    Log.i(TAG, String.format("dispatchStubActivity,[%s -> %s]",
targetClassName, stubActivity));
    intent.setClassName(mContext, stubActivity);
}
```

可以直接看最后一行，intent通过setClassName替换启动的目标Activity了！这个stubActivity是由 `mStubActivityInfo.getStubActivity(targetClassName, launchMode, themeObj)` 返回。

很明显，传入的参数launchMode、themeObj都是决定选择哪一个占坑类用的。

```
public String getStubActivity(String className, int launchMode,
Theme theme) {
    String stubActivity= mCachedStubActivity.get(className);
    if (stubActivity != null) {
        return stubActivity;
    }

    TypedArray array = theme.obtainStyledAttributes(new int[]{
            android.R.attr.windowIsTranslucent,
            android.R.attr.windowBackground
    });
    boolean windowIsTranslucent = array.getBoolean(0, false);
```

```
    array.recycle();
    if (Constants.DEBUG) {
        Log.d("StubActivityInfo", "getStubActivity, is transpare
nt theme ? " + windowIsTranslucent);
    }
    stubActivity = String.format(STUB_ACTIVITY_STANDARD, corePac
kage, usedStandardStubActivity);
    switch (launchMode) {
        case ActivityInfo.LAUNCH_MULTIPLE: {
            stubActivity = String.format(STUB_ACTIVITY_STANDARD,
 corePackage, usedStandardStubActivity);
            if (windowIsTranslucent) {
                stubActivity = String.format(STUB_ACTIVITY_STAND
ARD, corePackage, 2);
            }
            break;
        }
        case ActivityInfo.LAUNCH_SINGLE_TOP: {
            usedSingleTopStubActivity = usedSingleTopStubActivit
y % MAX_COUNT_SINGLETOP + 1;
            stubActivity = String.format(STUB_ACTIVITY_SINGLETOP
, corePackage, usedSingleTopStubActivity);
            break;
        }

       // 省略LAUNCH_SINGLE_TASK，LAUNCH_SINGLE_INSTANCE
    }

    mCachedStubActivity.put(className, stubActivity);
    return stubActivity;
}
```

可以看到主要就是根据launchMode去选择不同的占坑类。例如：

```
stubActivity = String.format(STUB_ACTIVITY_STANDARD, corePackage
, usedStandardStubActivity);
```

`STUB_ACTIVITY_STANDARD值为："%s.A$%d"`，corePackage值为`com.didi.virtualapk.core`，usedStandardStubActivity为数字值。

所以最终类名格式为： `com.didi.virtualapk.core.A$1`

再看一眼，CoreLibrary下的AndroidManifest中：

```
<activity android:name=".A$1" android:launchMode="standard"/>
<activity android:name=".A$2" android:launchMode="standard"
    android:theme="@android:style/Theme.Translucent" />

<!-- Stub Activities -->
<activity android:name=".B$1" android:launchMode="singleTop"/>
<activity android:name=".B$2" android:launchMode="singleTop"/>
<activity android:name=".B$3" android:launchMode="singleTop"/>
// 省略很多...    1234567891234567789
```

就完全明白了。

到这里就可以看到，替换我们启动的Activity为占坑Activity，将我们原本启动的包名，类名存储到了Intent中。

这样做只完成了一半，为什么这么说呢？

(2) 还原Activity

因为欺骗过了AMS，AMS执行完成后，最终要启动的不可能是占坑Activity，还应该是我们的启动的目标Activity呀。

这里需要知道Activity的启动流程：

AMS在处理完启动Activity后，会调用： `app.thread.scheduleLaunchActivity` ，这里的thread对应的server端未我们ActivityThread中的ApplicationThread对象(binder可以理解有一个client端和一个server端)，所以会调用 `ApplicationThread.scheduleLaunchActivity` 方法，在其内部会调用mH类的sendMessage方法，传递的标识为 `H.LAUNCH_ACTIVITY` ，进入调用到ActivityThread的handleLaunchActivity方法->ActivityThread#handleLaunchActivity->mInstrumentation.newActivity()。

> ps:这里流程不清楚没关系，暂时理解为最终会回调到Instrumentation的newActivity方法即可，细节可以自己去查看结合老罗的blog理解。

关键的来了，最终又到了Instrumentation的newActivity方法，还记得这个类我们已经改为VAInstrumentation啦：

直接看其newActivity方法：

```
@Override
public Activity newActivity(ClassLoader cl, String className, In
tent intent) throws InstantiationException, IllegalAccessExcepti
on, ClassNotFoundException {
    try {
        cl.loadClass(className);
    } catch (ClassNotFoundException e) {
        LoadedPlugin plugin = this.mPluginManager.getLoadedPlugi
n(intent);
        String targetClassName = PluginUtil.getTargetActivity(in
tent);

        if (targetClassName != null) {
            Activity activity = mBase.newActivity(plugin.getClas
sLoader(), targetClassName, intent);
            activity.setIntent(intent);

          // 省略兼容性处理代码
            return activity;
        }
    }

    return mBase.newActivity(cl, className, intent);
}
```

核心就是首先从intent中取出我们的目标Activity，然后通过plugin的ClassLoader去加载（还记得在加载插件时，会生成一个LoadedPlugin对象，其中会对应其初始化一个DexClassLoader）。

这样就完成了Activity的“偷梁换柱”。

还没完，接下来在 `callActivityOnCreate` 方法中：

```
 @Override
public void callActivityOnCreate(Activity activity, Bundle icicl
e) {
    final Intent intent = activity.getIntent();
    if (PluginUtil.isIntentFromPlugin(intent)) {
        Context base = activity.getBaseContext();
        try {
            LoadedPlugin plugin = this.mPluginManager.getLoadedP
lugin(intent);
            ReflectUtil.setField(base.getClass(), base, "mResour
ces", plugin.getResources());
            ReflectUtil.setField(ContextWrapper.class, activity,
"mBase", plugin.getPluginContext());
            ReflectUtil.setField(Activity.class, activity, "mApp
lication", plugin.getApplication());
            ReflectUtil.setFieldNoException(ContextThemeWrapper.
class, activity, "mBase", plugin.getPluginContext());

            // set screenOrientation
            ActivityInfo activityInfo = plugin.getActivityInfo(P
luginUtil.getComponent(intent));
            if (activityInfo.screenOrientation != ActivityInfo.S
CREEN_ORIENTATION_UNSPECIFIED) {
                activity.setRequestedOrientation(activityInfo.sc
reenOrientation);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }

    }

    mBase.callActivityOnCreate(activity, icicle);
}
```

设置了修改了mResources、mBase（Context）、mApplication对象。以及设置一些可动态设置的属性，这里仅设置了屏幕方向。

这里提一下，将mBase替换为PluginContext，可以修改Resources、AssetManager以及拦截相当多的操作。

看一眼代码就清楚了：

原本Activity的部分get操作

```
# ContextWrapper
@Override
public AssetManager getAssets() {
    return mBase.getAssets();
}

@Override
public Resources getResources()
{
    return mBase.getResources();
}

@Override
public PackageManager getPackageManager() {
    return mBase.getPackageManager();
}

@Override
public ContentResolver getContentResolver() {
    return mBase.getContentResolver();
}
```

直接替换为：

```
# PluginContext

@Override
public Resources getResources() {
    return this.mPlugin.getResources();
}

@Override
public AssetManager getAssets() {
    return this.mPlugin.getAssets();
}

@Override
public ContentResolver getContentResolver() {
    return new PluginContentResolver(getHostContext());
}
```

看得出来还是非常巧妙的。可以做的事情也非常多，后面对ContentProvider的描述也会提现出来。

好了，到此Activity就可以正常启动了。

下面看Service。

三、Service的支持

Service和Activity有点不同，显而易见的首先我们也会将要启动的Service类替换为占坑的Service类，但是有一点不同，在Standard模式下多次启动同一个占坑Activity会创建多个对象来对象我们的目标类。而Service多次启动只会调用onStartCommond方法，甚至常规多次调用bindService，seviceConn对象不变，甚至都不会多次回调bindService方法（多次调用可以通过给Intent设置不同Action解决）。

还有一点，最明显的差异是，Activity的生命周期是由用户交互决定的，而Service的声明周期是我们主动通过代码调用的。

也就是说，start、stop、bind、unbind都是我们显示调用的，所以我们可以拦截这几个方法，做一些事情。

Virtual Apk的做法，即将所有的操作进行拦截，都改为startService，然后统一在onStartCommond中分发。

下面看详细代码：

(1) hook IActivityManager

再次来到PluginManager，发下如下方法：

```
private void hookSystemServices() {
    try {
        Singleton<IActivityManager> defaultSingleton = (Singleton<IActivityManager>) ReflectUtil.getField(ActivityManagerNative.class, null, "gDefault");
        IActivityManager activityManagerProxy = ActivityManagerProxy.newInstance(this, defaultSingleton.get());

        // Hook IActivityManager from ActivityManagerNative
        ReflectUtil.setField(defaultSingleton.getClass().getSuperclass(), defaultSingleton, "mInstance", activityManagerProxy);

        if (defaultSingleton.get() == activityManagerProxy) {
            this.mActivityManager = activityManagerProxy;
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

首先拿到ActivityManagerNative中的gDefault对象，该对象返回的是一个 `Singleton<IActivityManager>` ,然后拿到其mInstance对象，即IActivityManager对象（可以理解为和AMS交互的binder的client对象）对象。

然后通过动态代理的方式，替换为了一个代理对象。

那么重点看对应的InvocationHandler对象即可，该代理对象调用的方法都会辗转到其invoke方法：

```
@Override
public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
    if ("startService".equals(method.getName())) {
        try {
            return startService(proxy, method, args);
        } catch (Throwable e) {
            Log.e(TAG, "Start service error", e);
        }
    } else if ("stopService".equals(method.getName())) {
        try {
            return stopService(proxy, method, args);
        } catch (Throwable e) {
            Log.e(TAG, "Stop Service error", e);
        }
    } else if ("stopServiceToken".equals(method.getName())) {
        try {
            return stopServiceToken(proxy, method, args);
        } catch (Throwable e) {
            Log.e(TAG, "Stop service token error", e);
        }
    }
    // 省略bindService，unbindService等方法
}
```

当我们调用startService时，跟进代码，可以发现调用流程为：

```
startService->startServiceCommon->ActivityManagerNative.getDefau
lt().startService
```

这个getDefault刚被我们hook，所以会被上述方法拦截，然后调用：`startService(proxy, method, args)`

```
private Object startService(Object proxy, Method method, Object[
] args) throws Throwable {
    IApplicationThread appThread = (IApplicationThread) args[0];
    Intent target = (Intent) args[1];
    ResolveInfo resolveInfo = this.mPluginManager.resolveService
(target, 0);
    if (null == resolveInfo || null == resolveInfo.serviceInfo)
{
        // is host service
        return method.invoke(this.mActivityManager, args);
    }

    return startDelegateServiceForTarget(target, resolveInfo.ser
viceInfo, null, RemoteService.EXTRA_COMMAND_START_SERVICE);
}
```

先不看代码，考虑下我们这里唯一要做的就是通过Intent保存关键数据，替换启动的Service类为占坑类。

所以直接看最后的方法：

```
private ComponentName startDelegateServiceForTarget(Intent targe
t,
                                                     ServiceInfo
serviceInfo,
                                                     Bundle extra
s, int command) {
    Intent wrapperIntent = wrapperTargetIntent(target, serviceIn
fo, extras, command);
    return mPluginManager.getHostContext().startService(wrapperI
ntent);
}
```

最后一行就是启动了，那么替换的操作应该在wrapperTargetIntent中完成：

```
private Intent wrapperTargetIntent(Intent target, ServiceInfo se
rviceInfo, Bundle extras, int command) {
    // fill in service with ComponentName
    target.setComponent(new ComponentName(serviceInfo.packageNam
e, serviceInfo.name));
    String pluginLocation = mPluginManager.getLoadedPlugin(targe
t.getComponent()).getLocation();

    // start delegate service to run plugin service inside
    boolean local = PluginUtil.isLocalService(serviceInfo);
    Class<? extends Service> delegate = local ? LocalService.cla
ss : RemoteService.class;
    Intent intent = new Intent();
    intent.setClass(mPluginManager.getHostContext(), delegate);
    intent.putExtra(RemoteService.EXTRA_TARGET, target);
    intent.putExtra(RemoteService.EXTRA_COMMAND, command);
    intent.putExtra(RemoteService.EXTRA_PLUGIN_LOCATION, pluginL
ocation);
    if (extras != null) {
        intent.putExtras(extras);
    }

    return intent;
}
```

果不其然，重新初始化了Intent，设置了目标类为LocalService（多进程时设置为RemoteService），然后将原本的Intent存储到 `EXTRA_TARGET` ，携带command为 `EXTRA_COMMAND_START_SERVICE` ，以及插件apk路径。

（2）代理分发

那么接下来代码就到了LocalService的onStartCommond中啦：

```
@Override
public int onStartCommand(Intent intent, int flags, int startId)
{
    // 省略一些代码...

```

```
    Intent target = intent.getParcelableExtra(EXTRA_TARGET);
    int command = intent.getIntExtra(EXTRA_COMMAND, 0);
    if (null == target || command <= 0) {
        return START_STICKY;
    }

    ComponentName component = target.getComponent();
    LoadedPlugin plugin = mPluginManager.getLoadedPlugin(compone
nt);

    switch (command) {
        case EXTRA_COMMAND_START_SERVICE: {
            ActivityThread mainThread = (ActivityThread)ReflectU
til.getActivityThread(getBaseContext());
            IApplicationThread appThread = mainThread.getApplica
tionThread();
            Service service;

            if (this.mPluginManager.getComponentsHandler().isSer
viceAvailable(component)) {
                service = this.mPluginManager.getComponentsHandl
er().getService(component);
            } else {
                try {
                    service = (Service) plugin.getClassLoader().
loadClass(component.getClassName()).newInstance();

                    Application app = plugin.getApplication();
                    IBinder token = appThread.asBinder();
                    Method attach = service.getClass().getMethod(
"attach", Context.class, ActivityThread.class, String.class, IBi
nder.class, Application.class, Object.class);
                    IActivityManager am = mPluginManager.getActi
vityManager();

                    attach.invoke(service, plugin.getPluginConte
xt(), mainThread, component.getClassName(), token, app, am);
                    service.onCreate();
                    this.mPluginManager.getComponentsHandler().r
ememberService(component, service);
```

```
                } catch (Throwable t) {
                    return START_STICKY;
                }
            }

            service.onStartCommand(target, 0, this.mPluginManage
r.getComponentsHandler().getServiceCounter(service).getAndIncrem
ent());
            break;
        }
        // 省略下面的代码
         case EXTRA_COMMAND_BIND_SERVICE:break;
         case EXTRA_COMMAND_STOP_SERVICE:break;
         case EXTRA_COMMAND_UNBIND_SERVICE:break;
}
```

这里代码很简单了，根据command类型，比如 `EXTRA_COMMAND_START_SERVICE` ，直接通过plugin的ClassLoader去load目标Service的class，然后反射创建实例。比较重要的是，Service创建好后，需要调用它的attach方法，这里凑够参数，然后反射调用即可，最后调用onCreate、onStartCommand收工。然后将其保存起来，stop的时候取出来调用其onDestroy即可。

bind、unbind以及stop的代码与上述基本一致，不在赘述。

唯一提醒的就是，刚才看到还hook了一个方法叫做： `stopServiceToken` ，该方法是什么时候用的呢？

主要有一些特殊的Service，比如IntentService，其stopSelf是由自身调用的，最终会调用 `mActivityManager.stopServiceToken` 方法，同样的中转为STOP操作即可。

四、BroadcastReceiver的支持

这个比较简单，直接解析Manifest后，静态转动态即可。

相关代码在LoadedPlugin的构造方法中：

```
for (PackageParser.Activity receiver : this.mPackage.receivers)
{
    receivers.put(receiver.getComponentName(), receiver.info);

    try {
        BroadcastReceiver br = BroadcastReceiver.class.cast(getC
lassLoader().loadClass(receiver.getComponentName().getClassName(
)).newInstance());
        for (PackageParser.ActivityIntentInfo aii : receiver.int
ents) {
            this.mHostContext.registerReceiver(br, aii);
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

可以看到解析到receiver信息后，直接通过pluginClassloader去loadClass拿到receiver对象，然后调用this.mHostContext.registerReceiver即可。

开心，最后一个了~

五、ContentProvider的支持

（1）hook IContentProvider

ContentProvider的支持依然是通过代理分发。

看一段CP使用的代码：

```
Cursor bookCursor = getContentResolver().query(bookUri, new Stri
ng[]{"_id", "name"}, null, null, null);
```

这里用到了PluginContext，在生成Activity、Service的时候，为其设置的Context都为PluginContext对象。

所以当你调用getContentResolver时，调用的为PluginContext的getContentResolver。

```
@Override
public ContentResolver getContentResolver() {
    return new PluginContentResolver(getHostContext());
}
```

返回的是一个PluginContentResolver对象，当我们调用query方法时，会辗转调用到 `ContentResolver.acquireUnstableProvider` 方法。该方法被PluginContentResolver中复写:

```
protected IContentProvider acquireUnstableProvider(Context conte
xt, String auth) {
    try {
        if (mPluginManager.resolveContentProvider(auth, 0) != nu
ll) {
            return mPluginManager.getIContentProvider();
        }

        return (IContentProvider) sAcquireUnstableProvider.invok
e(mBase, context, auth);
    } catch (Exception e) {
        e.printStackTrace();
    }

    return null;
}
```

如果调用的auth为插件apk中的provider，则直接返回 `mPluginManager.getIContentProvider()` 。

```
public synchronized IContentProvider getIContentProvider() {
    if (mIContentProvider == null) {
        hookIContentProviderAsNeeded();
    }

    return mIContentProvider;
}
```

咦，又看到一个hook方法：

```
private void hookIContentProviderAsNeeded() {
    Uri uri = Uri.parse(PluginContentResolver.getUri(mContext));
    mContext.getContentResolver().call(uri, "wakeup", null, null
);
    try {
        Field authority = null;
        Field mProvider = null;
        ActivityThread activityThread = (ActivityThread) Reflect
Util.getActivityThread(mContext);
        Map mProviderMap = (Map) ReflectUtil.getField(activityTh
read.getClass(), activityThread, "mProviderMap");
        Iterator iter = mProviderMap.entrySet().iterator();
        while (iter.hasNext()) {
            Map.Entry entry = (Map.Entry) iter.next();
            Object key = entry.getKey();
            Object val = entry.getValue();
            String auth;
            if (key instanceof String) {
                auth = (String) key;
            } else {
                if (authority == null) {
                    authority = key.getClass().getDeclaredField(
"authority");
                    authority.setAccessible(true);
                }
                auth = (String) authority.get(key);
            }
            if (auth.equals(PluginContentResolver.getAuthority(m
Context))) {
```

```
                if (mProvider == null) {
                    mProvider = val.getClass().getDeclaredField(
"mProvider");
                    mProvider.setAccessible(true);
                }
                IContentProvider rawProvider = (IContentProvider
) mProvider.get(val);
                IContentProvider proxy = IContentProviderProxy.n
ewInstance(mContext, rawProvider);
                mIContentProvider = proxy;
                Log.d(TAG, "hookIContentProvider succeed : " + m
IContentProvider);
                break;
            }
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

前两行比较重要，第一行是拿到了占坑的provider的uri，然后主动调用了其call方法。 如果你跟进去，会发现，其会调用acquireProvider->mMainThread.acquireProvider->ActivityManagerNative.getDefault().getContentProvider->installProvider。简单来说，其首先调用已经注册provider，得到返回的IContentProvider对象。

这个IContentProvider对象是在ActivityThread.installProvider方法中加入到mProviderMap中。

而ActivityThread对象又容易获取，mProviderMap又是它成员变量，那么也容易获取，所以上面的一大坨（除了前两行）代码，就为了拿到占坑的provider对应的IContentProvider对象。

然后通过动态代理的方式，进行了hook，关注InvocationHandler的实例IContentProviderProxy。

IContentProvider能干吗呢？其实就能拦截我们正常的query、insert、update、delete等操作。

拦截这些方法干嘛？

当然是修改uri啦，把用户调用的uri，替换为占坑provider的uri，再把原本的uri作为参数拼接在占坑provider的uri后面即可。

好了，直接看invoke方法：

```
@Override
public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
    Log.v(TAG, method.toGenericString() + " : " + Arrays.toStrin
g(args));
    wrapperUri(method, args);

    try {
        return method.invoke(mBase, args);
    } catch (InvocationTargetException e) {
        throw e.getTargetException();
    }
}
```

直接看wrapperUri

```
private void wrapperUri(Method method, Object[] args) {
    Uri uri = null;
    int index = 0;
    if (args != null) {
        for (int i = 0; i < args.length; i++) {
            if (args[i] instanceof Uri) {
                uri = (Uri) args[i];
                index = i;
                break;
            }
        }
    }

    // 省略部分代码

    PluginManager pluginManager = PluginManager.getInstance(mCon
text);
    ProviderInfo info = pluginManager.resolveContentProvider(uri
.getAuthority(), 0);
    if (info != null) {
        String pkg = info.packageName;
        LoadedPlugin plugin = pluginManager.getLoadedPlugin(pkg)
;
        String pluginUri = Uri.encode(uri.toString());
        StringBuilder builder = new StringBuilder(PluginContentR
esolver.getUri(mContext));
        builder.append("/?plugin=" + plugin.getLocation());
        builder.append("&pkg=" + pkg);
        builder.append("&uri=" + pluginUri);
        Uri wrapperUri = Uri.parse(builder.toString());
        if (method.getName().equals("call")) {
            bundleInCallMethod.putString(KEY_WRAPPER_URI, wrappe
rUri.toString());
        } else {
            args[index] = wrapperUri;
        }
    }
}
```

从参数中找到uri，往下看，搞了个StringBuilder首先加入占坑provider的uri，然后将目标uri，pkg,plugin等参数等拼接上去，替换到args中的uri，然后继续走原本的流程。

假设是query方法，应该就到达我们占坑provider的query方法啦。

（2）代理分发

占坑如下：

```
<provider
    android:name="com.didi.virtualapk.delegate.RemoteContentProv
ider"
    android:authorities="${applicationId}.VirtualAPK.Provider"
    android:process=":daemon" />
```

打开RemoteContentProvider，直接看query方法：

```
@Override
public Cursor query(Uri uri, String[] projection, String selecti
on,
                    String[] selectionArgs, String sortOrder) {

    ContentProvider provider = getContentProvider(uri);
    Uri pluginUri = Uri.parse(uri.getQueryParameter(KEY_URI));
    if (provider != null) {
        return provider.query(pluginUri, projection, selection,
selectionArgs, sortOrder);
    }

    return null;
}
```

可以看到通过传入的生成了一个新的provider,然后拿到目标uri，在直接调用provider.query传入目标uri即可。

那么这个provider实际上是这个代理类帮我们生成的：

```
private ContentProvider getContentProvider(final Uri uri) {
    final PluginManager pluginManager = PluginManager.getInstanc
e(getContext());
    Uri pluginUri = Uri.parse(uri.getQueryParameter(KEY_URI));
    final String auth = pluginUri.getAuthority();
    // 省略了缓存管理
    LoadedPlugin plugin = pluginManager.getLoadedPlugin(uri.getQ
ueryParameter(KEY_PKG));
    if (plugin == null) {
        try {
            pluginManager.loadPlugin(new File(uri.getQueryParame
ter(KEY_PLUGIN)));
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    final ProviderInfo providerInfo = pluginManager.resolveConte
ntProvider(auth, 0);
    if (providerInfo != null) {
        RunUtil.runOnUiThread(new Runnable() {
            @Override
            public void run() {
                try {
                    LoadedPlugin loadedPlugin = pluginManager.ge
tLoadedPlugin(uri.getQueryParameter(KEY_PKG));
                    ContentProvider contentProvider = (ContentPr
ovider) Class.forName(providerInfo.name).newInstance();
                    contentProvider.attachInfo(loadedPlugin.getP
luginContext(), providerInfo);
                    sCachedProviders.put(auth, contentProvider);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }, true);
        return sCachedProviders.get(auth);
    }
    return null;
```

```
}
```

很简单，取出原本的uri，拿到auth，在通过加载plugin得到providerInfo，反射生成provider对象，在调用其attachInfo方法即可。

其他的几个方法：insert、update、delete、call逻辑基本相同，就不赘述了。

感觉这里其实通过hook AMS的getContentProvider方法也能完成上述流程，感觉好像可以更彻底，不需要依赖PluginContext了。

六、总结

总结下，其实就是文初的内容，可以看到VritualApk大体方案如下：

- Activity：在宿主apk中提前占几个坑，然后通过“欺上瞒下”（这个词好像是360之前的ppt中提到）的方式，启动插件apk的Activity；因为要支持不同的launchMode以及一些特殊的属性，需要占多个坑。
- Service：通过代理Service的方式去分发；主进程和其他进程，VirtualAPK使用了两个代理Service。
- BroadcastReceiver：静态转动态。
- ContentProvider：通过一个代理Provider进行分发。

Android推送服务的几种实现方式

现实生活中, 推送服务就像订杂志一样, 只要留下你的地址, 杂志就能如期送到你手里, 可以认为每个人都有唯一的一个地址, 但在目前的网络上, 这是办不到的, 因为不是每个人都有一个唯一的地址, 服务器想要给我们推送一条消息, 必须知道我们的地址, 但服务器不知道我们在哪.

说到推送服务, 我所知道的实现方案有如下几种:

轮询

客户端定期询问服务器有没有新的消息, 这样服务器不用管客户端的地址是什么, 客户端来问, 直接告诉它就行.

这种方案最简单, 对于一些不追求实时性的客户端来说, 很适合, 只需要把时间间隔设定成几个小时取一次, 就能很方便的解决问题.

但对于即时通讯产品来说, 这种方案完全不能用. 假设即时通讯软件在网络畅通的情况下发送的消息要求对方10s内就能收到, 如果用轮询, 那么客户端要每隔5s连一次服务器, 如果在移动端, 手机的电量和流量很快就会被消耗殆尽.

SMS通知

这种方案在移动端是有可能的, 让客户端拦截手机短信, 服务器在有新消息时给用户的手机号发一条特殊的短信, 客户端拦截短信后发现是正常短信就放行, 如果是特殊短信就连接服务器取消息.

运营商不会配合, 用户也不会放心, 这方案普通公司玩不起.

长连接

这大概是目前情况下最佳的方案了, 客户端主动和服务器建立TCP长连接之后, 客户端定期向服务器发送心跳包, 有消息的时候, 服务器直接通过这个已经建立好的TCP连接通知客户端.

XMPP, MQTT等不算推送技术

在网上搜索资料的时候, 经常看见*XMPP*协议实现的*Android*推送和*MQTT*协议实现的*Android*推送, 我个人觉得这两种说法都怪怪的, XMPP和MQTT二者都是协议, 尽管我不清楚严格来讲这俩协议工作在哪一层, 但是绝对是在传输层之上的, 姑且认为他俩在TCP/IP四层模型的应用层吧, 闭口不提传输层的实现, 而是扯应用层, 这种说法真是令我费解, 所以我个人认为XMPP, MQTT等等不算推送技术.

> 关于为什么TCP/IP是四层模型, 感谢评论区指出, 对应的是 应用层, 传输层, 网络层, 网络接口层, 也有说法把网络接口层分成两层, 这样就有了五层, 因为TCP/IP是事实上的模型, 所以说法不一很正常, 主流说法是四层.

关于这个XMPP, 我想很多人都是参考Openfire和Smack那套东西, 我一年前尝试用aSmack和Openfire做IM, 不过那个时候什么都不懂, 做的东西很烂, 唯一懂的就是Openfire这东西相当老了, 我看有一些开源的推送解决方案都是在这套东西的基础上改的, 想想这工作量, 挺可怕的.

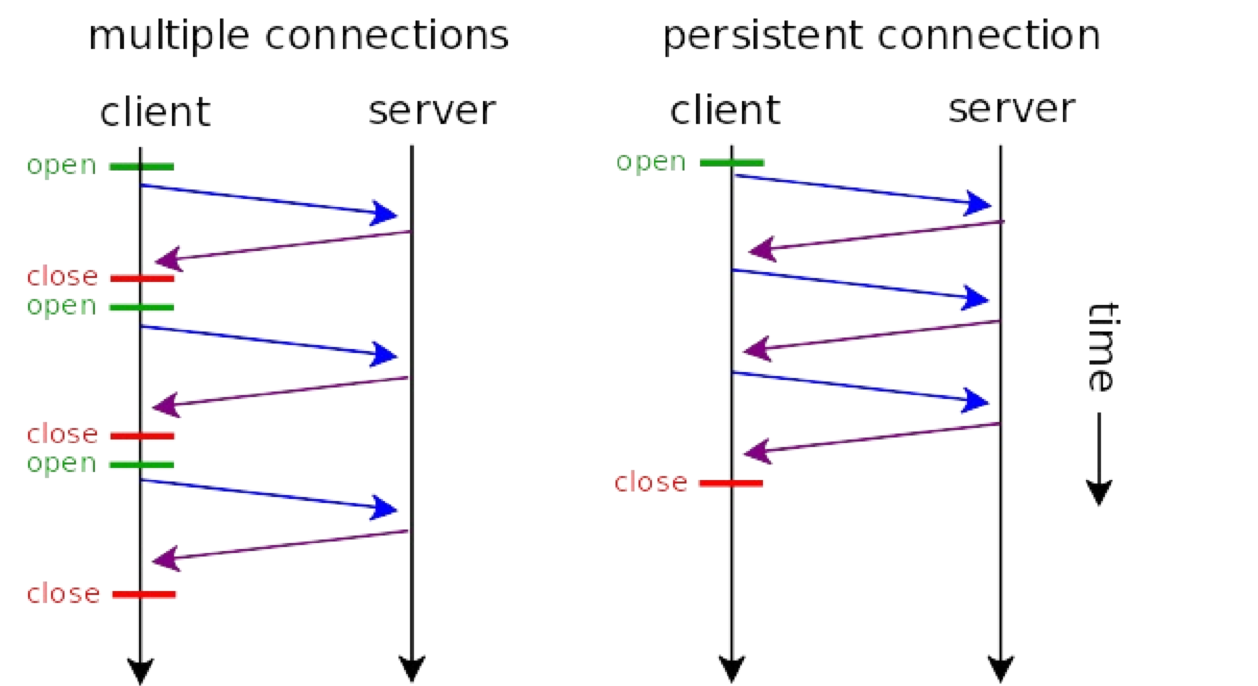
细说TCP长连接与心跳

长连接方案乍一听怪怪的, 什么是长连接? 定时发送心跳, 这和轮询有什么区别? 心跳是干什么的? 同样是定期和服务器沟通, 为什么长连接就比轮询更加优秀? 手机休眠了TCP连接不会断掉吗?

这是我在刚开始研究推送技术的时候的问题, 虽然有些还是没有很准确的答案, 但了解的大概可以分享一下, 有什么错误欢迎指出.

什么是长连接

先说短连接, 短连接是通讯双方有数据交互时就建立一个连接, 数据发送完成后，则断开此连接.



长连接就是大家建立连接之后，不主动断开. 双方互相发送数据，发完了也不主动断开连接，之后有需要发送的数据就继续通过这个连接发送.

TCP连接在默认的情况下就是所谓的长连接，也就是说连接双方都不主动关闭连接，这个连接就应该一直存在.

但是网络中的情况是复杂的，这个连接可能会被切断. 比如客户端到服务器的链路因为故障断了，或者服务器宕机了，或者是你家网线被人剪了，这些都是一些莫名其妙的导致连接被切断的因素，还有几种比较特殊的：

NAT超时

因为IPv4地址不足，或者我们想通过无线路由器上网，我们的设备可能会处在一个NAT设备的后面，生活中最常见的NAT设备是家用路由器.

NAT设备会在IP封包通过设备时修改源/目的IP地址. 对于家用路由器来说，使用的是网络地址端口转换(NAPT)，它不仅改IP，还修改TCP和UDP协议的端口号，这样就能让内网中的设备共用同一个外网IP. 举个例子，NAPT维护一个类似下表的NAT表

内网地址	外网地址
192.168.0.2:5566	120.132.92.21:9200
192.168.0.3:7788	120.132.92.21:9201
192.168.0.3:8888	120.132.92.21:9202

NAT设备会根据NAT表对出去和进来的数据做修改, 比如将 `192.168.0.3:8888` 发出去的封包改成 `120.132.92.21:9202` , 外部就认为他们是在和 `120.132.92.21:9202` 通信. 同时NAT设备会将 `120.132.92.21:9202` 收到的封包的IP和端口改成 `192.168.0.3:8888` , 再发给内网的主机, 这样内部和外部就能双向通信了, 但如果其中 `192.168.0.3:8888 == 120.132.92.21:9202` 这一映射因为某些原因被NAT设备淘汰了, 那么外部设备就无法直接与 `192.168.0.3:8888` 通信了.

我们的设备经常是处在NAT设备的后面, 比如在大学里的校园网, 查一下自己分配到的IP, 其实是内网IP, 表明我们在NAT设备后面, 如果我们在寝室再接个路由器, 那么我们发出的数据包会多经过一次NAT.

国内移动无线网络运营商在链路上一段时间内没有数据通讯后, 会淘汰NAT表中的对应项, 造成链路中断.

网络状态切换

手机网络和WIFI网络切换, 网络断开和连上等情况, 也会使长连接断开. 这里原因可能比较多, 但结果无非就是IP变了, 或者被系统通知连接断了.

DHCP的租期

> 目前测试发现安卓系统对DHCP的处理有Bug, DHCP租期到了不会主动续约并且会继续使用过期IP, 这个问题会造成TCP长连接偶然的断连.
>
> 引自Android微信智能心跳方案

心跳包的作用

网上很多文章介绍长连接的时候都说:

> 因为是长连接, 所以需要定期发送心跳包. 心跳包是用来通知服务器客户端当前状态.

提出这些说法的人其实自己也是一知半解. 这些说法其实都对, 但是没有答到点上. 就好像别人问: "你为什么要去食堂"? 这人回答: "检查自己还能不能找到食堂". 这个答案说不上错了, 但是其实这人是去食堂吃饭的, 证明自己认得路只是个附赠品.

明确一点, TCP长连接本质上不需要心跳包来维持, 大家可以试一试, 让两台电脑连上同一个wifi, 然后让其中一台做服务器, 另一台用一个普通的没有设置 `KeepAlive` 的 `Socket` 连上服务器, 只要两台电脑别断网, 路由器也别断电, DHCP正常续租, 就这么放着, 过几个小时再用其中一台电脑通过之前建立的TCP连接给另一台发消息, 另一台肯定能收到.

那为什么要有心跳包呢? 其实主要是为了防止上面提到的**NAT**超时, 既然一些NAT设备判断是否淘汰NAT映射的依据是一定时间没有数据, 那么客户端就主动发一个数据.

当然, 如果仅仅是为了防止NAT超时, 可以让服务器来发送心跳包给客户端, 不过这样做有个弊病就是, 万一连接断了, 服务器就再也联系不上客户端了. 所以心跳包必须由客户端发送, 客户端发现连接断了, 还可以尝试重连服务器.

所以心跳包的主要作用是防止NAT超时, 其次是探测连接是否断开.

链路断开, 没有写操作的TCP连接是感知不到的, 除非这个时候发送数据给服务器, 造成写超时, 否则TCP连接不会知道断开了. 主动kill掉一方的进程, 另一方会关闭TCP连接, 是系统代进程给服务器发的FIN. TCP连接就是这样, 只有明确的收到对方发来的关闭连接的消息(收到RST也会关闭, 大家都懂), 或者自己意识到发生了写超时, 否则它认为连接还存在.

心跳包的时间间隔

既然心跳包的主要作用是防止NAT超时, 那么这个间隔就大有文章了.

发送心跳包势必要先唤醒设备, 然后才能发送, 如果唤醒设备过于频繁, 或者直接导致设备无法休眠, 会大量消耗电量, 而且移动网络下进行网络通信, 比在wifi下耗电得多. 所以这个心跳包的时间间隔应该尽量的长, 最理想的情况就是根本没有NAT超时, 比如刚才我说的两台在同一个wifi下的电脑, 完全不需要心跳包. 这也就是网上常说的长连接, 慢心跳.

现实是残酷的, 根据网上的一些说法, 中移动2/3G下, NAT超时时间为5分钟, 中国电信3G则大于28分钟, 理想的情况下, 客户端应当以略小于NAT超时时间的间隔来发送心跳包.

wifi下, NAT超时时间都会比较长, 据说宽带的网关一般没有空闲释放机制, GCM有些时候在wifi下的心跳比在移动网络下的心跳要快, 可能是因为wifi下联网通信耗费的电量比移动网络下小.

关于如何让心跳间隔逼近NAT超时的间隔，同时自动适应NAT超时间隔的变化，可以参看Android微信智能心跳方案.

服务器如何处理心跳包

如果客户端心跳间隔是固定的，那么服务器在连接闲置超过这个时间还没收到心跳时，可以认为对方掉线，关闭连接. 如果客户端心跳会动态改变，如上节提到的微信心跳方案，应当设置一个最大值，超过这个最大值才认为对方掉线. 还有一种情况就是服务器通过TCP连接主动给客户端发消息出现写超时，可以直接认为对方掉线.

这个就需要具体业务具体分析了，也许还有更优的策略，这里就不写了.

心跳包和轮询的区别

心跳包和轮询看起来类似，都是客户端主动联系服务器，但是区别很大.

- 轮询是为了获取数据，而心跳是为了保活TCP连接.
- 轮询得越频繁，获取数据就越及时，心跳的频繁与否和数据是否及时没有直接关系
- 轮询比心跳能耗更高，因为一次轮询需要经过TCP三次握手，四次挥手，单次心跳不需要建立和拆除TCP连接.

TCP唤醒Android

这部分内容我只知道结论，不知道具体的知识 大家有没有想过，手机的短信功能和微信的功能差不多，为什么微信会比短信耗电这么多？当然不是因为短信一条0.1元. 手机短信是通过什么获取推送的呢？下面这段出处不明的话也许可以给大家启示

> 首先Android手机有两个处理器, 一个叫Application Processor(AP), 一个叫Baseband Processor(BP). AP是ARM架构的处理器，用于运行Android系统; BP用于运行实时操作系统(RTOS), 通讯协议栈运行于BP的RTOS之上. 非通话时间, BP的能耗基本上在5mA左右，而AP只要处于非休眠状态, 能耗至少在50mA以上, 执行图形运算时会更高. 另外LCD工作时功耗在100mA左右, WIFI也在100mA左右. 一般手机待机时, AP, LCD, WIFI均进入休眠状态, 这时Android中应用程序的代码也会停止执行.
>
> Android为了确保应用程序中关键代码的正确执行, 提供了Wake Lock的API, 使得应用程序有权限通过代码阻止AP进入休眠状态. 但如果不领会Android设计者的意图而滥用Wake Lock API, 为了自身程序在后台的正常工作而长时间阻止AP进入休眠状态, 就会成为待机电池杀手.
>
> 完全没必要担心AP休眠会导致收不到消息推送. 通讯协议栈运行于BP，一旦收到数据包, BP会将AP唤醒, 唤醒的时间足够AP执行代码完成对收到的数据包的处理过程. 其它的如Connectivity事件触发时AP同样会被唤醒. 那么唯一的问题就是程序如何执行向服务器发送心跳包的逻辑. 你显然不能靠AP来做心跳计时. Android提供的Alarm Manager就是来解决这个问题的. Alarm应该是BP计时(或其它某个带石英钟的芯片，不太确定，但绝对不是AP), 触发时唤醒AP执行程序代码. 那么Wake Lock API有啥用呢? 比如心跳包从请求到应答, 比如断线重连重新登陆这些关键逻辑的执行过程, 就需要Wake Lock来保护. 而一旦一个关键逻辑执行成功, 就应该立即释放掉Wake Lock了. 两次心跳请求间隔5到10分钟, 基本不会怎么耗电. 除非网络不稳定. 频繁断线重连, 那种情况办法不多.

上面所说的通信协议, 我猜应该是无线资源控制协议(Radio Resource Control), RRC应该工作在OSI参考模型中的第三层网络层, 而TCP, UDP工作在第四层传输层, 上文说的BP, 应该就是手机中的基带, 也有叫Radio的, 我有点搞不清楚Radio怎么翻译. Google在Optimizing Downloads for Efficient Network Access中提到了一个叫Radio State Machine的东西, 我翻译成无线电波状态机, 也不知道正确的翻译是什么.

移动网络下, 每一个TCP连接底层都应该是有RRC连接, 而RRC连接会唤醒基带, 基带会唤醒CPU处理TCP数据, 这是我个人的理解.

至于wifi下如何工作, 我暂时没有找到资料.

上面说了这么多, 其实意思就是TCP数据包能唤醒手机. 至于UDP, 我不确定.

而推送中最重要的部分就是让手机尽量休眠，只有在服务器需要它处理数据时才唤醒它，这正好符合我们的要求.

移动网络下的耗电

Google在Optimizing Downloads for Efficient Network Access中提到了一个叫Radio State Machine的东西.

mobile radio state machine

说的应该就是基带的工作状态，在Radio Standby下几乎不耗电，但是一旦有需要处理的事情，比如手机里某个app要访问网络(从上一节可以推测: 收到RRC指令也会导致唤醒)，就会进入到Radio Full Power中，由Standby转为Full Power这一唤醒过程很耗电，Full Power下基带空闲后5s进入Radio Low Power，如果又空闲12s才进入Standby. 主要的意思就是不要频繁的唤醒基带去请求网络，因为只要一唤醒，就至少会让基带在Full Power下工作5s，在Low Power下工作12s，而且唤醒过程很耗电. 所以在移动网络下，心跳需要尽量的慢才好，不过以当前这种情况，想慢下来几乎不可能.

不过这也带来另外一个问题，假如手机里有10个应用，每个应用都发送心跳包，每个应用的服务器都可能唤醒手机，那手机还休不休眠了?

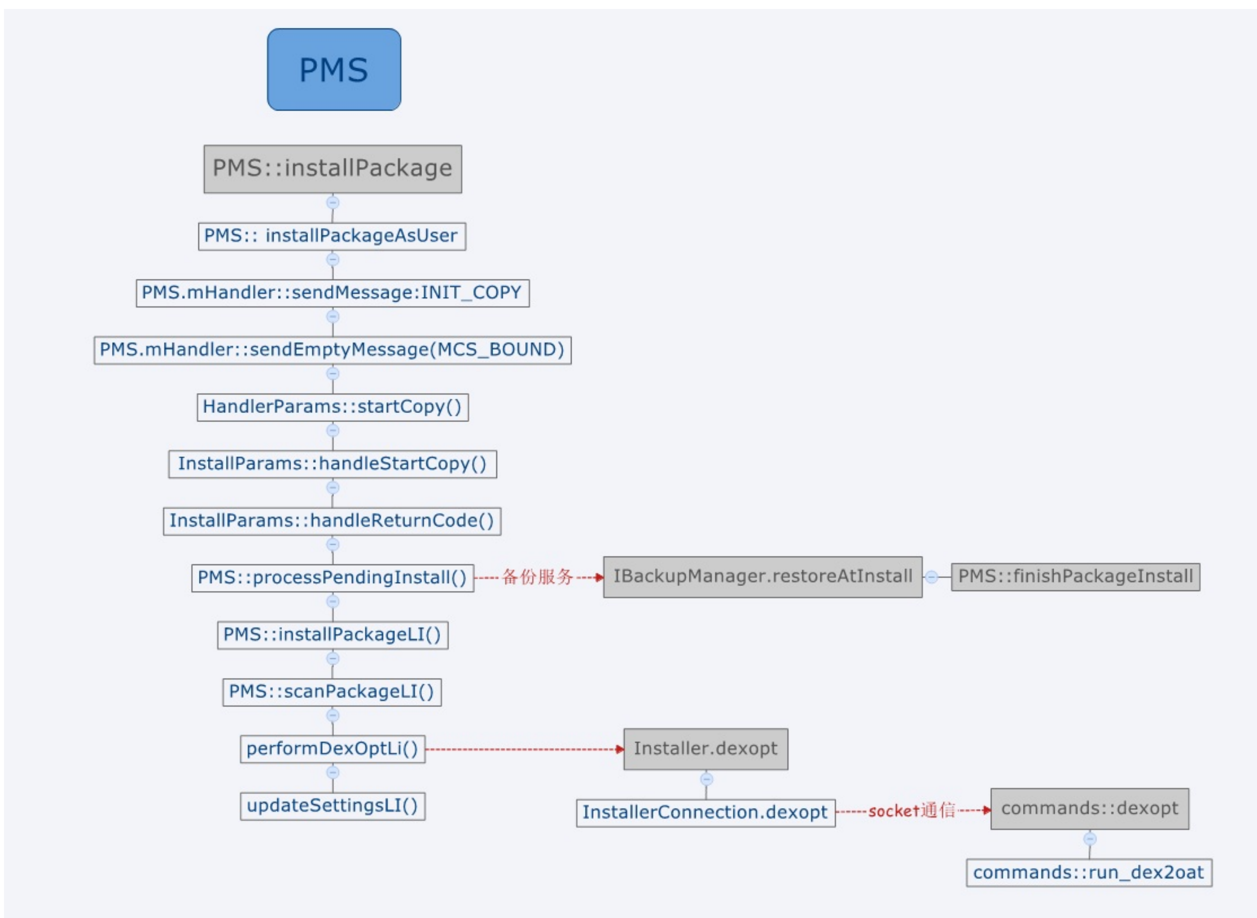
该文件修订时间： 2018-05-12 01:44:07

Apk安装的主要步骤:

为了学习这个过程,真的是陷入了pms的源码很久,也看了很多前人的博文,才算是有了些思路,所以此处先把主要步骤列出来,后面再慢慢分析细节。

1. 将apk文件复制到 `data/app` 目录
2. 解析apk信息
3. dexopt操作
4. 更新权限信息
5. 完成安装,发送 `Intent.ACTION_PACKAGE_ADDED` 广播

下面将具体步骤列张图出来:



由图可见安装过程中流转的步骤还是比较多的,下面具体分析

1. 将apk文件copy至 `data/app` 目录

1.1 installPackageAsUser

```
 mContext.enforceCallingOrSelfPermission(android.Manifest.permis
sion.INSTALL_PACKAGES, null);

        final int callingUid = Binder.getCallingUid();
        ...
        ...

        if ((callingUid == Process.SHELL_UID) || (callingUid ==
Process.ROOT_UID)) {
            installFlags |= PackageManager.INSTALL_FROM_ADB;

        } else {
            // Caller holds INSTALL_PACKAGES permission, so we'r
e less strict
            // about installerPackageName.

            installFlags &= ~PackageManager.INSTALL_FROM_ADB;
            installFlags &= ~PackageManager.INSTALL_ALL_USERS;
        }

        UserHandle user;
        if ((installFlags & PackageManager.INSTALL_ALL_USERS) !=
0) {
            user = UserHandle.ALL;
        } else {
            user = new UserHandle(userId);
        }

        verificationParams.setInstallerUid(callingUid);

        final File originFile = new File(originPath);
        final OriginInfo origin = OriginInfo.fromUntrustedFile(o
riginFile);

        final Message msg = mHandler.obtainMessage(INIT_COPY);
        msg.obj = new InstallParams(origin, observer, installFla
gs,
                installerPackageName, verificationParams, user,
packageAbiOverride);
```

```
        mHandler.sendMessage(msg);
```

这个方法主要是判断安装来源,包括adb,shell,all_user,然后向PMS的 `mHandler` 发送 `INIT_COPY` 的消息,这个 `mHandler` 运行在一个 `HandlerThread` 中。

1.2 handleMessage(INIT_COPY)&handleMessage(MCS_BOUND)

```
case INIT_COPY:{
                    HandlerParams params = (HandlerParams) msg.o
bj;
                    int idx = mPendingInstalls.size();
                    if (DEBUG_INSTALL) Slog.i(TAG, "init_copy id
x=" + idx + ": " + params);
                    // If a bind was already initiated we dont r
eally
                    // need to do anything. The pending install
                    // will be processed later on.
                    if (!mBound) {
                        // If this is the only one pending we mi
ght
                        // have to bind to the service again.
                        if (!connectToService()) {
                            Slog.e(TAG, "Failed to bind to media
 container service");
                            params.serviceError();
                            return;
                        } else {
                            // Once we bind to the service, the
first
                            // pending request will be processed.

                            mPendingInstalls.add(idx, params);
                        }
                    } else {
                        mPendingInstalls.add(idx, params);
```

```
                        // Already bound to the service. Just ma
ke
                        // sure we trigger off processing the fi
rst request.
                        if (idx == 0) {
                            mHandler.sendEmptyMessage(MCS_BOUND)
;
                        }
                    }
}
case MCS_BOUND:{
...
...
 HandlerParams params = mPendingInstalls.get(0);
                        if (params != null) {
                            if (params.startCopy()) {
                                // We are done...  look for more
 work or to
                                // go idle.
                                if (DEBUG_SD_INSTALL) Log.i(TAG,
                                        "Checking for more work
or unbind...");
                                // Delete pending install
                                if (mPendingInstalls.size() > 0)
 {
                                    mPendingInstalls.remove(0);
                                }
                                if (mPendingInstalls.size() == 0
) {
                                    if (mBound) {
                                        if (DEBUG_SD_INSTALL) Lo
g.i(TAG,
                                                "Posting delayed
 MCS_UNBIND");
                                        removeMessages(MCS_UNBIN
D);
                                        Message ubmsg = obtainMe
ssage(MCS_UNBIND);
                                        // Unbind after a little
 delay, to avoid
```

```
                                            // continual thrashing.
                                            sendMessageDelayed(ubmsg
, 10000);
                                        }
...
...
}
```

INIT_COPY主要是确保`DefaultContainerService`已bound,`DefaultContainerService`是一个应用服务，具体负责实现APK等相关资源文件在内部或外部存储器上的存储工作。而`MCS_BOUND`中则执行了`params.startCopy()`这句,也是最关键的开始copy文件。

1.3 HandlerParams.startCopy

```
  final boolean startCopy() {
            boolean res;
            try {
                if (DEBUG_INSTALL) Slog.i(TAG, "startCopy " + mU
ser + ": " + this);

                if (++mRetries > MAX_RETRIES) {
                    Slog.w(TAG, "Failed to invoke remote methods
 on default container service. Giving up");
                    mHandler.sendEmptyMessage(MCS_GIVE_UP);
                    handleServiceError();
                    return false;
                } else {
                    handleStartCopy();
                    res = true;
                }
            } catch (RemoteException e) {
                if (DEBUG_INSTALL) Slog.i(TAG, "Posting install
MCS_RECONNECT");
                mHandler.sendEmptyMessage(MCS_RECONNECT);
                res = false;
            }
            handleReturnCode();
            return res;
        }
```

该方法中除了检查重试次数外只是简单的调用了 `handleStartCopy()` 及 `handleReturnCode()` 方法.

1.4 handleStartCopy()

这个方法内容非常多,下面只列出些核心部分

```
 public void handleStartCopy() throws RemoteException {
           int ret = PackageManager.INSTALL_SUCCEEDED;
            ...
            ...
            final boolean onSd = (installFlags & PackageManager.
```

```
INSTALL_EXTERNAL) != 0;
            final boolean onInt = (installFlags & PackageManager
.INSTALL_INTERNAL) != 0;

            PackageInfoLite pkgLite = null;

            if (onInt && onSd) {
                // Check if both bits are set.
                Slog.w(TAG, "Conflicting flags specified for ins
talling on both internal and external");
                ret = PackageManager.INSTALL_FAILED_INVALID_INST
ALL_LOCATION;
            } else {
                pkgLite = mContainerService.getMinimalPackageInf
o(origin.resolvedPath, installFlags,
                        packageAbiOverride);

                /*
                 * If we have too little free space, try to free
 cache
                 * before giving up.
                 */
                if (!origin.staged && pkgLite.recommendedInstall
Location
                        == PackageHelper.RECOMMEND_FAILED_INSUFF
ICIENT_STORAGE) {
                    final StorageManager storage = StorageManage
r.from(mContext);
                    final long lowThreshold = storage.getStorage
LowBytes(
                            Environment.getDataDirectory());

                    final long sizeBytes = mContainerService.cal
culateInstalledSize(
                            origin.resolvedPath, isForwardLocked
(), packageAbiOverride);

                    if (mInstaller.freeCache(sizeBytes + lowThre
shold) >= 0) {
                        pkgLite = mContainerService.getMinimalPa
```

```
ckageInfo(origin.resolvedPath,
                                installFlags, packageAbiOverride
);
                    }
                }
            }
            ...
            ...
                     * No package verification is enabled, so im
mediately start
                     * the remote call to initiate copy using te
mporary file.
                     */
                    ret = args.copyApk(mContainerService, true);

            }
            mRet = ret;
        }
```

`handleStartCopy` 的核心就是copyApk,其他的都是些存储空间检查,权限检查等等安全校验

2 .解析apk信息

完成apk copy到 `data/app` 目录的操作后,下一步就到了 `handleReturnCode` ,这个方法又跳转到 `processPendingInstall()` 方法,下面先来看看 `processPendingInstall()` 方法:

2.1 processPendingInstall()

```
    private void processPendingInstall(final InstallArgs args, f
inal int currentStatus) {
        // Queue up an async operation since the package install
ation may take a little while.
        mHandler.post(new Runnable() {
            public void run() {
                mHandler.removeCallbacks(this);
```

```
                 // Result object to be returned
                PackageInstalledInfo res = new PackageInstalledI
nfo();
                res.returnCode = currentStatus;
                res.uid = -1;
                res.pkg = null;
                res.removedInfo = new PackageRemovedInfo();
                if (res.returnCode == PackageManager.INSTALL_SUC
CEEDED) {
                    args.doPreInstall(res.returnCode);
                    synchronized (mInstallLock) {
                        installPackageLI(args, res); //1.安装
                    }
                    args.doPostInstall(res.returnCode, res.uid);
                }

                // A restore should be performed at this point i
f (a) the install
                // succeeded, (b) the operation is not an update
, and (c) the new
                // package has not opted out of backup participa
tion.
                final boolean update = res.removedInfo.removedPa
ckage != null;
                final int flags = (res.pkg == null) ? 0 : res.pk
g.applicationInfo.flags;
                boolean doRestore = !update
                        && ((flags & ApplicationInfo.FLAG_ALLOW_
BACKUP) != 0);

                // Set up the post-install work request bookkeep
ing.  This will be used
                // and cleaned up by the post-install event hand
ling regardless of whether
                // there's a restore pass performed.  Token valu
es are >= 1.
                int token;
                if (mNextInstallToken < 0) mNextInstallToken = 1
;
                token = mNextInstallToken++;
```

```
                PostInstallData data = new PostInstallData(args,
 res);
                mRunningInstalls.put(token, data);
                if (DEBUG_INSTALL) Log.v(TAG, "+ starting restor
e round-trip " + token);

                if (res.returnCode == PackageManager.INSTALL_SUC
CEEDED && doRestore) {
                    // Pass responsibility to the Backup Manager
.  It will perform a
                    // restore if appropriate, then pass respons
ibility back to the
                    // Package Manager to run the post-install o
bserver callbacks
                    // and broadcasts.
                    IBackupManager bm = IBackupManager.Stub.asIn
terface(
                            ServiceManager.getService(Context.BA
CKUP_SERVICE));
                    if (bm != null) {
                        if (DEBUG_INSTALL) Log.v(TAG, "token " +
 token
                                + " to BM for possible restore")
;
                        try {
                            bm.restoreAtInstall(res.pkg.applicat
ionInfo.packageName, token); //2.调用backup服务
                        } catch (RemoteException e) {
                            // can't happen; the backup manager
is local
                        } catch (Exception e) {
                            Slog.e(TAG, "Exception trying to enq
ueue restore", e);
                            doRestore = false;
                        }
                    } else {
                        Slog.e(TAG, "Backup Manager not found!")
;
                        doRestore = false;
```

```
                    }
                }

                if (!doRestore) {
                    // No restore possible, or the Backup Manage
r was mysteriously not
                    // available -- just fire the post-install w
ork request directly.
                    if (DEBUG_INSTALL) Log.v(TAG, "No restore -
queue post-install for " + token);
                    Message msg = mHandler.obtainMessage(POST_IN
STALL, token, 0);
                    mHandler.sendMessage(msg);
                }
            }
        });
    }
```

这个方法有几个关键步骤,一是 `installPackageLI(args, res);` ,这个方法具体执行了解析package和后续操作,而再 `installPackageLI(args, res);` 执行完毕后会走到 `bm.restoreAtInstall(res.pkg.applicationInfo.packageName, token);` ,会调用backupservice的 `restoreAtInstall` 方法,而 `restoreAtInstall` 方法最终又会调用 `PMS` 的 `finishPackageInstall()` 方法,完成安装。

2.2 installPackageLI(args, res)

```
private void installPackageLI(InstallArgs args, PackageInstalled
Info res) {
        final int installFlags = args.installFlags;
        String installerPackageName = args.installerPackageName;
        File tmpPackageFile = new File(args.getCodePath());
        boolean forwardLocked = ((installFlags & PackageManager.
INSTALL_FORWARD_LOCK) != 0);
        boolean onSd = ((installFlags & PackageManager.INSTALL_E
XTERNAL) != 0);
        boolean replace = false;
        final int scanFlags = SCAN_NEW_INSTALL | SCAN_FORCE_DEX
```

```
| SCAN_UPDATE_SIGNATURE;
        // Result object to be returned
        res.returnCode = PackageManager.INSTALL_SUCCEEDED;

        if (DEBUG_INSTALL) Slog.d(TAG, "installPackageLI: path="
 + tmpPackageFile);
        // Retrieve PackageSettings and parse package
        final int parseFlags = mDefParseFlags | PackageParser.PA
RSE_CHATTY
                | (forwardLocked ? PackageParser.PARSE_FORWARD_L
OCK : 0)
                | (onSd ? PackageParser.PARSE_ON_SDCARD : 0);
        PackageParser pp = new PackageParser();
        pp.setSeparateProcesses(mSeparateProcesses);
        pp.setDisplayMetrics(mMetrics);

        final PackageParser.Package pkg;
        try {
            pkg = pp.parsePackage(tmpPackageFile, parseFlags);
        } catch (PackageParserException e) {
            res.setError("Failed parse during installPackageLI",
 e);
            return;
        }

        // Mark that we have an install time CPU ABI override.
        pkg.cpuAbiOverride = args.abiOverride;

        String pkgName = res.name = pkg.packageName;
        if ((pkg.applicationInfo.flags&ApplicationInfo.FLAG_TEST
_ONLY) != 0) {
            if ((installFlags & PackageManager.INSTALL_ALLOW_TES
T) == 0) {
                res.setError(INSTALL_FAILED_TEST_ONLY, "installP
ackageLI");
                return;
            }
        }

        try {
```

```
            pp.collectCertificates(pkg, parseFlags);
            pp.collectManifestDigest(pkg);
        } catch (PackageParserException e) {
            res.setError("Failed collect during installPackageLI"
, e);
            return;
        }

        /* If the installer passed in a manifest digest, compare
 it now. */
        if (args.manifestDigest != null) {
            if (DEBUG_INSTALL) {
                final String parsedManifest = pkg.manifestDigest
 == null ? "null"
                            : pkg.manifestDigest.toString();
                Slog.d(TAG, "Comparing manifests: " + args.manif
estDigest.toString() + " vs. "
                            + parsedManifest);
            }

            if (!args.manifestDigest.equals(pkg.manifestDigest))
 {
                res.setError(INSTALL_FAILED_PACKAGE_CHANGED, "Ma
nifest digest changed");
                return;
            }
        } else if (DEBUG_INSTALL) {
            final String parsedManifest = pkg.manifestDigest ==
null
                        ? "null" : pkg.manifestDigest.toString();
            Slog.d(TAG, "manifestDigest was not present, but par
ser got: " + parsedManifest);
        }

        // Get rid of all references to package scan path via pa
rser.
        pp = null;
        String oldCodePath = null;
        boolean systemApp = false;
        synchronized (mPackages) {
```

```
            // Check whether the newly-scanned package wants to
define an already-defined perm
            int N = pkg.permissions.size();
            for (int i = N-1; i >= 0; i--) {
                PackageParser.Permission perm = pkg.permissions.
get(i);
                BasePermission bp = mSettings.mPermissions.get(p
erm.info.name);
                if (bp != null) {
                    // If the defining package is signed with ou
r cert, it's okay.  This
                    // also includes the "updating the same pack
age" case, of course.
                    // "updating same package" could also involv
e key-rotation.
                    final boolean sigsOk;
                    if (!bp.sourcePackage.equals(pkg.packageName
)
                            || !(bp.packageSetting instanceof Pa
ckageSetting)
                            || !bp.packageSetting.keySetData.isU
singUpgradeKeySets()
                            || ((PackageSetting) bp.packageSetti
ng).sharedUser != null) {
                        sigsOk = compareSignatures(bp.packageSet
ting.signatures.mSignatures,
                                pkg.mSignatures) == PackageManag
er.SIGNATURE_MATCH;
                    } else {
                        sigsOk = checkUpgradeKeySetLP((PackageSe
tting) bp.packageSetting, pkg);
                    }
                    if (!sigsOk) {
                        // If the owning package is the system i
tself, we log but allow
                        // install to proceed; we fail the insta
ll on all other permission
                        // redefinitions.
                        if (!bp.sourcePackage.equals("android"))
 {
```

```
                            res.setError(INSTALL_FAILED_DUPLICAT
E_PERMISSION, "Package "
                                    + pkg.packageName + " attemp
ting to redeclare permission "
                                    + perm.info.name + " already
 owned by " + bp.sourcePackage);
                            res.origPermission = perm.info.name;
                            res.origPackage = bp.sourcePackage;
                            return;
                        } else {
                            Slog.w(TAG, "Package " + pkg.package
Name
                                    + " attempting to redeclare
system permission "
                                    + perm.info.name + "; ignori
ng new declaration");
                            pkg.permissions.remove(i);
                        }
                    }
                }
            }

            // Check if installing already existing package
            if ((installFlags & PackageManager.INSTALL_REPLACE_E
XISTING) != 0) {
                String oldName = mSettings.mRenamedPackages.get(
pkgName);
                if (pkg.mOriginalPackages != null
                        && pkg.mOriginalPackages.contains(oldNam
e)

                        && mPackages.containsKey(oldName)) {
                    // This package is derived from an original
package,
                    // and this device has been updating from th
at original
                    // name.  We must continue using the origina
l name, so
                    // rename the new package here.
                    pkg.setPackageName(oldName);
                    pkgName = pkg.packageName;
```

```
                    replace = true;
                    if (DEBUG_INSTALL) Slog.d(TAG, "Replacing ex
isting renamed package: oldName="
                            + oldName + " pkgName=" + pkgName);
                } else if (mPackages.containsKey(pkgName)) {
                    // This package, under its official name, al
ready exists
                    // on the device; we should replace it.
                    replace = true;
                    if (DEBUG_INSTALL) Slog.d(TAG, "Replace exis
ting pacakge: " + pkgName);
                }
            }
            PackageSetting ps = mSettings.mPackages.get(pkgName)
;
            if (ps != null) {
                if (DEBUG_INSTALL) Slog.d(TAG, "Existing package
: " + ps);
                oldCodePath = mSettings.mPackages.get(pkgName).c
odePathString;
                if (ps.pkg != null && ps.pkg.applicationInfo !=
null) {
                    systemApp = (ps.pkg.applicationInfo.flags &
                            ApplicationInfo.FLAG_SYSTEM) != 0;
                }
                res.origUsers = ps.queryInstalledUsers(sUserMana
ger.getUserIds(), true);
            }
        }

        if (systemApp && onSd) {
            // Disable updates to system apps on sdcard
            res.setError(INSTALL_FAILED_INVALID_INSTALL_LOCATION
,
                    "Cannot install updates to system apps on sd
card");
            return;
        }

        if (!args.doRename(res.returnCode, pkg, oldCodePath)) {
```

```
            res.setError(INSTALL_FAILED_INSUFFICIENT_STORAGE, "F
ailed rename");
            return;
        }

        if (replace) {
            replacePackageLI(pkg, parseFlags, scanFlags | SCAN_R
EPLACING, args.user,
                    installerPackageName, res);
        } else {
            installNewPackageLI(pkg, parseFlags, scanFlags | SCA
N_DELETE_DATA_ON_FAILURES,
                    args.user, installerPackageName, res);
        }
        synchronized (mPackages) {
            final PackageSetting ps = mSettings.mPackages.get(pk
gName);
            if (ps != null) {
                res.newUsers = ps.queryInstalledUsers(sUserManag
er.getUserIds(), true);
            }
        }
    }
```

这个方法先是解析了package包,然后做了大量签名和权限校验的工作,最终会走到

```
  if (replace) {
            replacePackageLI(pkg, parseFlags, scanFlags | SCAN_R
EPLACING, args.user,
                    installerPackageName, res);
        } else {
            installNewPackageLI(pkg, parseFlags, scanFlags | SCA
N_DELETE_DATA_ON_FAILURES,
                    args.user, installerPackageName, res);
        }
```

这两个方法分别是覆盖安装和安装新应用对应的具体执行.我们来看看 `installNewPackageLI()`

2.3 installNewPackageLI()

```
    private void installNewPackageLI(PackageParser.Package pkg,
            int parseFlags, int scanFlags, UserHandle user,
            String installerPackageName, PackageInstalledInfo re
s) {
        // Remember this for later, in case we need to rollback
this install
        String pkgName = pkg.packageName;

        if (DEBUG_INSTALL) Slog.d(TAG, "installNewPackageLI: " +
 pkg);
        boolean dataDirExists = getDataPathForPackage(pkg.packag
eName, 0).exists();
        synchronized(mPackages) {
            if (mSettings.mRenamedPackages.containsKey(pkgName))
 {
                // A package with the same name is already insta
lled, though
                // it has been renamed to an older name.  The pa
ckage we
                // are trying to install should be installed as
an update to
                // the existing one, but that has not been reque
sted, so bail.
                res.setError(INSTALL_FAILED_ALREADY_EXISTS, "Att
empt to re-install " + pkgName
                        + " without first uninstalling package r
unning as "
                        + mSettings.mRenamedPackages.get(pkgName
));
                return;
            }
            if (mPackages.containsKey(pkgName)) {
                // Don't allow installation over an existing pac
kage with the same name.
                res.setError(INSTALL_FAILED_ALREADY_EXISTS, "Att
empt to re-install " + pkgName
                        + " without first uninstalling.");
```

```
                return;
            }
        }

        try {
            PackageParser.Package newPackage = scanPackageLI(pkg
, parseFlags, scanFlags,
                    System.currentTimeMillis(), user);

            updateSettingsLI(newPackage, installerPackageName, n
ull, null, res);
            // delete the partially installed application. the d
ata directory will have to be
            // restored if it was already existing
            if (res.returnCode != PackageManager.INSTALL_SUCCEED
ED) {
                // remove package from internal structures.  Not
e that we want deletePackageX to
                // delete the package data and cache directories
 that it created in
                // scanPackageLocked, unless those directories e
xisted before we even tried to
                // install.
                deletePackageLI(pkgName, UserHandle.ALL, false,
null, null,
                        dataDirExists ? PackageManager.DELETE_KE
EP_DATA : 0,
                                res.removedInfo, true);
            }

        } catch (PackageManagerException e) {
            res.setError("Package couldn't be installed in " + p
kg.codePath, e);
        }
    }
```

这个方法核心的步骤有两个:

- `PackageParser.Package newPackage = scanPackageLI(pkg, parseFlags, scanFlags,System.currentTimeMillis(), user);`

- `updateSettingsLI(newPackage, installerPackageName, null, null, res);`

scanPackageLI负责安装,而updateSettingLI则是完成安装后的设置信息更新

2.4 scanPackageLI()

scanPackageLI()方法主要逻辑是由 `scanPackageDirtyLI()` 实现的, `scanPackageDirtyLI()` 实在太长了,此处就不列出了,主要说下,这个方法实现了以下操作:

- 设置系统App的一些参数
- 校验签名
- 解析app的provider,校验是否与已有的provider冲突
- 32/64位abi的一些设置
- 四大组件的解析，注册

`scanPackageDirtyLI()` 里面的操作确实是太多了，并不止这几点。如需更详细的信息还请查看源码。

另一方面，这个方法里,会调用到 `performDexOptLI()` ,其会去执行 `dexopt操作` 。

3. dexopt操作

Apk文件其实只是一个归档zip压缩包,而我们编写的代码最终都编译成了 `.dex` 文件,但为了提高运行性能,android系统并不会直接执行 `.dex` ,而是会在安装过程中执行 `dexopt` 操作来优化 `.dex` 文件,最终android系统执行的时优化后的'odex'文件(注意:这个odex文件的后缀也是.dex,其路径在 `data/dalvik-cache`)。对于 `dalvik` 虚拟机, `dexopt` 就是优化操作,而对于 `art` 虚拟机, `dexopt` 执行的则是 `dex2oat` 操作,既将 `.dex` 文件翻译成 `oat` 文件。关于 `art` 和 `dex2oat` 的更多信息请看后文。

这里我们先来看看 `PMS` 的 `dexopt` 操作:

3.1 performDexOptLI()

这个方法的核心是

```
final int ret = mInstaller.dexopt(path, sharedGid, !isForwardLoc
ked(pkg), pkg.packageName, dexCodeInstructionSet, vmSafeMode);
```

其作用就是调用 `PMS` 的 `mInstaller` 成员变量的 `dexopt` 操作。

3.2 Installer.dexopt

Installer类的dexopt方法又调用 `InstallerConnection` 类的 `dexopt` 方法,来看看这个方法:

```
public int dexopt(String apkPath, int uid, boolean isPublic, Str
ing pkgName,
            String instructionSet, boolean vmSafeMode) {
        StringBuilder builder = new StringBuilder("dexopt");
        builder.append(' ');
        builder.append(apkPath);
        builder.append(' ');
        builder.append(uid);
        builder.append(isPublic ? " 1" : " 0");
        builder.append(' ');
        builder.append(pkgName);
        builder.append(' ');
        builder.append(instructionSet);
        builder.append(' ');
        builder.append(vmSafeMode ? " 1" : " 0");
        return execute(builder.toString());
    }
 public synchronized String transact(String cmd) {
        if (!connect()) {
            Slog.e(TAG, "connection failed");
            return "-1";
        }

        if (!writeCommand(cmd)) {
            /*
             * If installd died and restarted in the background
(unlikely but
             * possible) we'll fail on the next write (this one)
```

```
. Try to
             * reconnect and write the command one more time bef
ore giving up.
             */
            Slog.e(TAG, "write command failed? reconnect!");
            if (!connect() || !writeCommand(cmd)) {
                return "-1";
            }
        }
        if (LOCAL_DEBUG) {
            Slog.i(TAG, "send: '" + cmd + "'");
        }

        final int replyLength = readReply();
        if (replyLength > 0) {
            String s = new String(buf, 0, replyLength);
            if (LOCAL_DEBUG) {
                Slog.i(TAG, "recv: '" + s + "'");
            }
            return s;
        } else {
            if (LOCAL_DEBUG) {
                Slog.i(TAG, "fail");
            }
            return "-1";
        }
    }

    public int execute(String cmd) {
        String res = transact(cmd);
        try {
            return Integer.parseInt(res);
        } catch (NumberFormatException ex) {
            return -1;
        }
    }
  private boolean connect() {
        if (mSocket != null) {
            return true;
        }
```

```
        Slog.i(TAG, "connecting...");
        try {
            mSocket = new LocalSocket();

            LocalSocketAddress address = new LocalSocketAddress(
"installd",
                    LocalSocketAddress.Namespace.RESERVED);

            mSocket.connect(address);

            mIn = mSocket.getInputStream();
            mOut = mSocket.getOutputStream();
        } catch (IOException ex) {
            disconnect();
            return false;
        }
        return true;
    }
```

由上面的几个方法可以知道,最终 `dexopt` 操作是通过**socket**的方式来跨进程通知守护进程 `installd` ,由其去执行 `dexopt` 操作。

3.3 commands::dexopt()

最终守护进程 `installd` 会调用 `Commands.c` 文件(位于 `/source/framework/native/cmds/installd`)的 `dexopt` 方法。

```
int dexopt(const char *apk_path, uid_t uid, bool is_public,
           const char *pkgname, const char *instruction_set,
           bool vm_safe_mode, bool is_patchoat)
{
    struct utimbuf ut;
    struct stat input_stat, dex_stat;
    char out_path[PKG_PATH_MAX];
    char persist_sys_dalvik_vm_lib[PROPERTY_VALUE_MAX];
    char *end;
    const char *input_file;
    char in_odex_path[PKG_PATH_MAX];
    int res, input_fd=-1, out_fd=-1;
```

```
    ...
    ...

    pid_t pid;
    pid = fork();
    if (pid == 0) {
        /* child -- drop privileges before continuing */
        if (setgid(uid) != 0) {
            ALOGE("setgid(%d) failed in installd during dexopt\n"
, uid);
            exit(64);
        }
        if (setuid(uid) != 0) {
            ALOGE("setuid(%d) failed in installd during dexopt\n"
, uid);
            exit(65);
        }
        // drop capabilities
        struct __user_cap_header_struct capheader;
        struct __user_cap_data_struct capdata[2];
        memset(&capheader, 0, sizeof(capheader));
        memset(&capdata, 0, sizeof(capdata));
        capheader.version = _LINUX_CAPABILITY_VERSION_3;
        if (capset(&capheader, &capdata[0]) < 0) {
            ALOGE("capset failed: %s\n", strerror(errno));
            exit(66);
        }
        if (set_sched_policy(0, SP_BACKGROUND) < 0) {
            ALOGE("set_sched_policy failed: %s\n", strerror(errn
o));
            exit(70);
        }
        if (flock(out_fd, LOCK_EX | LOCK_NB) != 0) {
            ALOGE("flock(%s) failed: %s\n", out_path, strerror(e
rrno));
            exit(67);
        }

        if (strncmp(persist_sys_dalvik_vm_lib, "libdvm", 6) == 0
```

```
) {
            run_dexopt(input_fd, out_fd, input_file, out_path);
        } else if (strncmp(persist_sys_dalvik_vm_lib, "libart", 6
) == 0) {
            if (is_patchoat) {
                run_patchoat(input_fd, out_fd, input_file, out_p
ath, pkgname, instruction_set);
            } else {
                run_dex2oat(input_fd, out_fd, input_file, out_pa
th, pkgname, instruction_set,
                            vm_safe_mode);
            }
        } else {
            exit(69);   /* Unexpected persist.sys.dalvik.vm.lib
value */
        }
        exit(68);   /* only get here on exec failure */
    } else {
        res = wait_child(pid);
        if (res == 0) {
            ALOGV("DexInv: --- END '%s' (success) ---\n", input_
file);
        } else {
            ALOGE("DexInv: --- END '%s' --- status=0x%04x, proce
ss failed\n", input_file, res);
            goto fail;
        }
    }

    ut.actime = input_stat.st_atime;
    ut.modtime = input_stat.st_mtime;
    utime(out_path, &ut);

    close(out_fd);
    close(input_fd);
    return 0;

fail:
    if (out_fd >= 0) {
        close(out_fd);
```

```
            unlink(out_path);
        }
        if (input_fd >= 0) {
            close(input_fd);
        }
        return -1;
    }
```

由上面的代码可以发现, `installd` 在做了些操作后, `fork` 出了一个新的进程,根据虚拟机的类型为 `libdvm` 或 `libart` 分别执行 `run_dexopt` 或 `run_dex2oat` (如果为is_patchoat,则是 `run_patchoat`)操作。

4. 更新权限信息

`dexopt` 操作执行完后, `installNewPackageLI()` 方法就会走到 `updateSettingsLI()` 来更新设置信息,而更新设置信息主要是权限信息,所以直接来看 `updatePermissionsLPw()` ;

4.1 updatePermissionsLPw

```
private void updatePermissionsLPw(String changingPkg,
            PackageParser.Package pkgInfo, int flags) {
        // Make sure there are no dangling permission trees.
        Iterator<BasePermission> it = mSettings.mPermissionTrees
.values().iterator();
        while (it.hasNext()) {
            final BasePermission bp = it.next();
            if (bp.packageSetting == null) {
                // We may not yet have parsed the package, so ju
st see if
                // we still know about its settings.
                bp.packageSetting = mSettings.mPackages.get(bp.s
ourcePackage);
            }
            if (bp.packageSetting == null) {
                Slog.w(TAG, "Removing dangling permission tree: "
 + bp.name
```

```
                        + " from package " + bp.sourcePackage);
                it.remove();
            } else if (changingPkg != null && changingPkg.equals
(bp.sourcePackage)) {
                if (pkgInfo == null || !hasPermission(pkgInfo, b
p.name)) {
                    Slog.i(TAG, "Removing old permission tree: "
 + bp.name
                            + " from package " + bp.sourcePackag
e);
                    flags |= UPDATE_PERMISSIONS_ALL;
                    it.remove();
                }
            }
        }

        // Make sure all dynamic permissions have been assigned
to a package,
        // and make sure there are no dangling permissions.
        it = mSettings.mPermissions.values().iterator();
        while (it.hasNext()) {
            final BasePermission bp = it.next();
            if (bp.type == BasePermission.TYPE_DYNAMIC) {
                if (DEBUG_SETTINGS) Log.v(TAG, "Dynamic permissi
on: name="
                        + bp.name + " pkg=" + bp.sourcePackage
                        + " info=" + bp.pendingInfo);
                if (bp.packageSetting == null && bp.pendingInfo
!= null) {
                    final BasePermission tree = findPermissionTr
eeLP(bp.name);
                    if (tree != null && tree.perm != null) {
                        bp.packageSetting = tree.packageSetting;
                        bp.perm = new PackageParser.Permission(t
ree.perm.owner,
                                new PermissionInfo(bp.pendingInf
o));
                        bp.perm.info.packageName = tree.perm.inf
o.packageName;
                        bp.perm.info.name = bp.name;
```

```
                        bp.uid = tree.uid;
                    }
                }
            }
            if (bp.packageSetting == null) {
                // We may not yet have parsed the package, so ju
st see if
                // we still know about its settings.
                bp.packageSetting = mSettings.mPackages.get(bp.s
ourcePackage);
            }
            if (bp.packageSetting == null) {
                Slog.w(TAG, "Removing dangling permission: " + b
p.name
                        + " from package " + bp.sourcePackage);
                it.remove();
            } else if (changingPkg != null && changingPkg.equals
(bp.sourcePackage)) {
                if (pkgInfo == null || !hasPermission(pkgInfo, b
p.name)) {
                    Slog.i(TAG, "Removing old permission: " + bp
.name
                            + " from package " + bp.sourcePackag
e);
                    flags |= UPDATE_PERMISSIONS_ALL;
                    it.remove();
                }
            }
        }

        // Now update the permissions for all packages, in parti
cular
        // replace the granted permissions of the system package
s.
        if ((flags&UPDATE_PERMISSIONS_ALL) != 0) {
            for (PackageParser.Package pkg : mPackages.values())
 {
                if (pkg != pkgInfo) {
                    grantPermissionsLPw(pkg, (flags&UPDATE_PERMI
SSIONS_REPLACE_ALL) != 0,
```

```
                            changingPkg);
                }
            }
        }

        if (pkgInfo != null) {
            grantPermissionsLPw(pkgInfo, (flags&UPDATE_PERMISSIO
NS_REPLACE_PKG) != 0, changingPkg);
        }
    }


    private void grantPermissionsLPw(PackageParser.Package pkg,
boolean replace,
            String packageOfInterest) {
        final PackageSetting ps = (PackageSetting) pkg.mExtras;
        if (ps == null) {
            return;
        }
        final GrantedPermissions gp = ps.sharedUser != null ? ps
.sharedUser : ps;
        HashSet<String> origPermissions = gp.grantedPermissions;
        boolean changedPermission = false;

        if (replace) {
            ps.permissionsFixed = false;
            if (gp == ps) {
                origPermissions = new HashSet<String>(gp.granted
Permissions);
                gp.grantedPermissions.clear();
                gp.gids = mGlobalGids;
            }
        }

        if (gp.gids == null) {
            gp.gids = mGlobalGids;
        }

        final int N = pkg.requestedPermissions.size();
        for (int i=0; i<N; i++) {
```

```
            final String name = pkg.requestedPermissions.get(i);
            final boolean required = pkg.requestedPermissionsReq
uired.get(i);
            final BasePermission bp = mSettings.mPermissions.get
(name);
            if (DEBUG_INSTALL) {
                if (gp != ps) {
                    Log.i(TAG, "Package " + pkg.packageName + "
checking " + name + ": " + bp);
                }
            }

            if (bp == null || bp.packageSetting == null) {
                if (packageOfInterest == null || packageOfIntere
st.equals(pkg.packageName)) {
                    Slog.w(TAG, "Unknown permission " + name
                            + " in package " + pkg.packageName);
                }
                continue;
            }

            final String perm = bp.name;
            boolean allowed;
            boolean allowedSig = false;
            if ((bp.protectionLevel&PermissionInfo.PROTECTION_FL
AG_APPOP) != 0) {
                // Keep track of app op permissions.
                ArraySet<String> pkgs = mAppOpPermissionPackages
.get(bp.name);
                if (pkgs == null) {
                    pkgs = new ArraySet<>();
                    mAppOpPermissionPackages.put(bp.name, pkgs);
                }
                pkgs.add(pkg.packageName);
            }
            final int level = bp.protectionLevel & PermissionInf
o.PROTECTION_MASK_BASE;
            if (level == PermissionInfo.PROTECTION_NORMAL
                    || level == PermissionInfo.PROTECTION_DANGER
OUS) {
```

```
                // We grant a normal or dangerous permission if
any of the following
                // are true:
                // 1) The permission is required
                // 2) The permission is optional, but was grante
d in the past
                // 3) The permission is optional, but was reques
ted by an
                //    app in /system (not /data)
                //
                // Otherwise, reject the permission.
                allowed = (required || origPermissions.contains(
perm)
                        || (isSystemApp(ps) && !isUpdatedSystemA
pp(ps)));
            } else if (bp.packageSetting == null) {
                // This permission is invalid; skip it.
                allowed = false;
            } else if (level == PermissionInfo.PROTECTION_SIGNAT
URE) {
                allowed = grantSignaturePermission(perm, pkg, bp
, origPermissions);
                if (allowed) {
                    allowedSig = true;
                }
            } else {
                allowed = false;
            }
            if (DEBUG_INSTALL) {
                if (gp != ps) {
                    Log.i(TAG, "Package " + pkg.packageName + "
granting " + perm);
                }
            }
            if (allowed) {
                if (!isSystemApp(ps) && ps.permissionsFixed) {
                    // If this is an existing, non-system packag
e, then
                    // we can't add any new permissions to it.
                    if (!allowedSig && !gp.grantedPermissions.co
```

```
ntains(perm)) {
                        // Except...  if this is a permission th
at was added
                        // to the platform (note: need to only d
o this when
                        // updating the platform).
                        allowed = isNewPlatformPermissionForPack
age(perm, pkg);
                    }
                }
                if (allowed) {
                    if (!gp.grantedPermissions.contains(perm)) {
                        changedPermission = true;
                        gp.grantedPermissions.add(perm);
                        gp.gids = appendInts(gp.gids, bp.gids);
                    } else if (!ps.haveGids) {
                        gp.gids = appendInts(gp.gids, bp.gids);
                    }
                } else {
                    if (packageOfInterest == null || packageOfIn
terest.equals(pkg.packageName)) {
                        Slog.w(TAG, "Not granting permission " +
 perm
                                + " to package " + pkg.packageNa
me
                                + " because it was previously in
stalled without");
                    }
                }
            } else {
                if (gp.grantedPermissions.remove(perm)) {
                    changedPermission = true;
                    gp.gids = removeInts(gp.gids, bp.gids);
                    Slog.i(TAG, "Un-granting permission " + perm
                            + " from package " + pkg.packageName
                            + " (protectionLevel=" + bp.protecti
onLevel
                            + " flags=0x" + Integer.toHexString(
pkg.applicationInfo.flags)
                            + ")");
```

```
                } else if ((bp.protectionLevel&PermissionInfo.PR
OTECTION_FLAG_APPOP) == 0) {
                    // Don't print warning for app op permission
s, since it is fine for them
                    // not to be granted, there is a UI for the
user to decide.
                    if (packageOfInterest == null || packageOfIn
terest.equals(pkg.packageName)) {
                        Slog.w(TAG, "Not granting permission " +
 perm
                                + " to package " + pkg.packageNa
me
                                + " (protectionLevel=" + bp.prot
ectionLevel
                                + " flags=0x" + Integer.toHexStr
ing(pkg.applicationInfo.flags)
                                + ")");
                    }
                }
            }
        }

        if ((changedPermission || replace) && !ps.permissionsFix
ed &&
                !isSystemApp(ps) || isUpdatedSystemApp(ps)){
            // This is the first that we have heard about this p
ackage, so the
            // permissions we have now selected are fixed until
explicitly
            // changed.
            ps.permissionsFixed = true;
        }
        ps.haveGids = true;
    }
```

由上面两个方法可以看到,在apk的安装时 `PMS` 会将该app的所有权限都记录下来并更新到 `PMS` 的 `mAppOpPermissionPackages` 成员变量里面,并判定是否授予该app请求的权限。

4.2 完成安装

还记得前面说过的在 `processPendingInstall` 方法在执行 `installPackageLi` 后会执行以下语句

```
 if (res.returnCode == PackageManager.INSTALL_SUCCEEDED && doRes
tore) {
                    // Pass responsibility to the Backup Manager
.  It will perform a
                    // restore if appropriate, then pass respons
ibility back to the
                    // Package Manager to run the post-install o
bserver callbacks
                    // and broadcasts.
                    IBackupManager bm = IBackupManager.Stub.asIn
terface(
                            ServiceManager.getService(Context.BA
CKUP_SERVICE));
                    if (bm != null) {
                        if (DEBUG_INSTALL) Log.v(TAG, "token " +
 token
                                + " to BM for possible restore")
;
                        try {
                            bm.restoreAtInstall(res.pkg.applicat
ionInfo.packageName, token);
                        } catch (RemoteException e) {
                            // can't happen; the backup manager
is local
                        } catch (Exception e) {
                            Slog.e(TAG, "Exception trying to enq
ueue restore", e);
                            doRestore = false;
                        }
                    } else {
                        Slog.e(TAG, "Backup Manager not found!")
;
                        doRestore = false;
                    }
                }
```

我也不是很清楚为什么系统会调用 `IBackupManager` 的 `restoreAtInstall` 方法,不过发现在 `BackupManagerService` 的 `restoreAtInstall` 方法中会有以下代码:

```
...
 if (skip) {
            // Auto-restore disabled or no way to attempt a rest
ore; just tell the Package
            // Manager to proceed with the post-install handling
 for this package.
            if (DEBUG) Slog.v(TAG, "Finishing install immediatel
y");
            try {
                mPackageManagerBinder.finishPackageInstall(token
);
            } catch (RemoteException e) { /* can't happen */ }
        }
...
```

最终 `restoreAtInstall` 方法又会调用 `PMS` 的 `finishPackageInstall` 方法,而此方法最终会发送 `Intent.ACTION_PACKAGE_ADDED` 广播，apk的安装就到到此结束了。

从底部弹出的选择框


14:33
100
我
15764230137
我的收藏
我的优惠券
检查更新
已经是最新版本
意见反馈
关于我们
清除本地缓存
14.5MB
拍照
选择
取消



使用popupWindow完成:

定义popupWindow类

```
/**
 * 选择照片的PopupWindow
 * Created by chenlijin on 2016/4/12.
 */
public class SelectPicPopupWindow extends PopupWindow implements
```

```
View.OnTouchListener, View.OnKeyListener {
    private Context mContext;
    private View rootView;

    public SelectPicPopupWindow(Context context) {
        mContext = context;
        LayoutInflater inflater = LayoutInflater.from(context);
        rootView = inflater.inflate(R.layout.popupwindow_selectp
ic, null);
        setContentView(rootView);
        ButterKnife.bind(this, rootView);
        //设置高度和宽度。
        this.setHeight(ViewGroup.LayoutParams.WRAP_CONTENT);
        this.setWidth(ViewGroup.LayoutParams.MATCH_PARENT);
        this.setFocusable(true);

        //设置动画效果
        this.setAnimationStyle(R.style.mypopwindow_anim_style);

        //当单击Back键或者其他地方使其消失、需要设置这个属性。
        rootView.setOnTouchListener(this);
        rootView.setOnKeyListener(this);
        rootView.setFocusable(true);
        rootView.setFocusableInTouchMode(true);

        //实例化一个ColorDrawable颜色为半透明
        ColorDrawable dw = new ColorDrawable(0xb0000000);
        //设置SelectPicPopupWindow弹出窗体的背景
        this.setBackgroundDrawable(dw);
    }


    //点击外部popup消失
    @Override
    public boolean onTouch(View v, MotionEvent event) {
        int height = rootView.findViewById(R.id.linearlayout_win
dow).getTop();
        int y = (int) event.getY();
        if (event.getAction() == MotionEvent.ACTION_UP) {
            if (y < height) {
```

```
                dismiss();
            }
        }
        return true;
    }

    //点back键消失
    @Override
    public boolean onKey(View v, int keyCode, KeyEvent event) {
        if (keyCode == KeyEvent.KEYCODE_BACK && this.isShowing()
) {
            this.dismiss();
            return true;
        }
        return false;
    }


    @OnClick({R.id.button_take_photo, R.id.button_select_pic, R.
id.button_cancal})
    public void onClick(View view) {
        switch (view.getId()) {
            case R.id.button_take_photo:
                listener.onClickTakePhoto();
                this.dismiss();
                break;
            case R.id.button_select_pic:
                listener.onClickSelectPic();
                this.dismiss();
                break;
            case R.id.button_cancal:
                this.dismiss();
                break;
        }
    }

    private OnWindowItemClickListener listener;

    public void setOnWindowItemClickListener(OnWindowItemClickLi
stener listener) {
```

```
        this.listener = listener;
    }

    public interface OnWindowItemClickListener {
        void onClickTakePhoto();

        void onClickSelectPic();
    }
}
```

自定义Style

```
<style name="MyPopup" parent="android:style/Theme.Dialog">
        <item name="android:windowFrame">@null</item>
        <item name="android:windowNoTitle">true</item>
        <item name="android:windowBackground">@color/popup</item>

        <item name="android:windowIsFloating">true</item>
        <item name="android:windowContentOverlay">@null</item>
    </style>
```

定义进入和退出的动画:

```
进入:
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android">
    <translate
        android:duration="200"
        android:fromYDelta="100.0%"
        android:toYDelta="0.0"/>
</set>
退出
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android">
    <translate
        android:duration="200"
        android:fromYDelta="0.0"
        android:toYDelta="100.0%"/>
</set>
动画的style
<style name="mypopwindow_anim_style">
        <item name="android:windowEnterAnimation">@anim/popup_in
</item>
        <!-- 指定显示的动画xml -->
        <item name="android:windowExitAnimation">@anim/popup_out
</item>
        <!-- 指定消失的动画xml -->
    </style>
```

在指定的位置显示

```
//显示窗口
window.showAtLocation(MainActivity.this.findViewById(R.id.main),
 Gravity.BOTTOM|Gravity.CENTER_HORIZONTAL, 0, 0); //设置layout在P
opupWindow中显示的位置
```

使用Dialog完成:

定义style

```
<!--自定义布局的dialog-->
    <style name="MyDialog" parent="android:style/Theme.Dialog">
        <!-- 背景颜色及透明程度 -->
        <item name="android:windowBackground">@android:color/tra
nsparent</item>
        <!-- 是否有标题 -->
        <item name="android:windowNoTitle">true</item>
        <!-- 是否浮现在activity之上,会造成macth_parent失效-->
        <item name="android:windowIsFloating">false</item>
        <!-- 是否模糊 -->
        <item name="android:backgroundDimEnabled">true</item>
        <item name="android:windowFrame">@null</item>
    </style>
```

动画：和popupwindow一致

自定义Dialog:

```
/**
 * 选择图片对话框
 * Created by chenlijin on 2016/4/12.
 */
public class SelectPicDialog extends Dialog {
    public SelectPicDialog(Context context, int themeResId) {
        super(context, themeResId);
    }

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.dialog_select_pic);
        ButterKnife.bind(this);

    }

    @OnClick({R.id.linearlayout_out,R.id.textview_take_photo, R.
id.textview_select_photo, R.id.textview_cancal})
```

```
    public void onClick(View view) {
        switch (view.getId()) {
            case R.id.textview_take_photo:
                if(listener!=null){
                    listener.onClickTakePhoto();
                }
                this.cancel();
                break;
            case R.id.textview_select_photo:
                if(listener!=null){
                    listener.onClickSelectPic();
                }
                this.cancel();
                break;
            case R.id.linearlayout_out:
            case R.id.textview_cancal:
                this.cancel();
                break;
        }
    }

    private OnWindowItemClickListener listener;

    public void setOnWindowItemClickListener(OnWindowItemClickLi
stener listener) {
        this.listener = listener;
    }

    public interface OnWindowItemClickListener {
        void onClickTakePhoto();
        void onClickSelectPic();
    }
}
```

在Activity中调用:

```
SelectPicDialog dialog = new SelectPicDialog(mContext,R.style.MyDialog);
        Window window = dialog.getWindow();
        window.setGravity(Gravity.BOTTOM);  //此处可以设置dialog显示的位置
        window.setWindowAnimations(R.style.mypopwindow_anim_style);  //添加动画
        dialog.show();
        dialog.setOnWindowItemClickListener(new SelectPicDialog.OnWindowItemClickListener(){

            @Override
            public void onClickTakePhoto() {
                startActivityForResult(createCameraIntent(), CREATE_CAMERA);   //选择拍照
            }

            @Override
            public void onClickSelectPic() {
                startActivityForResult(createPickIntent(), CREATE_PICK);   //选择启用系统的选择图片
            }
        });
```

详细的区别

（1）Popupwindow在显示之前一定要设置宽高，Dialog无此限制。

（2）Popupwindow默认不会响应物理键盘的back，除非显示设置了popup.setFocusable(true);而在点击back的时候，Dialog会消失。

（3）Popupwindow不会给页面其他的部分添加蒙层，而Dialog会。

（4）Popupwindow没有标题，Dialog默认有标题，可以通过dialog.requestWindowFeature(Window.FEATURE_NO_TITLE);取消标题

（5）二者显示的时候都要设置Gravity。如果不设置，Dialog默认是Gravity.CENTER。

（6）二者都有默认的背景，都可以通过setBackgroundDrawable(new ColorDrawable(android.R.color.transparent));去掉。

其中最本质的差别就是：AlertDialog是非阻塞式对话框：AlertDialog弹出时，后台还可以做事情；而PopupWindow是阻塞式对话框：PopupWindow弹出时，程序会等待，在PopupWindow退出前，程序一直等待，只有当我们调用了dismiss方法的后，PopupWindow退出，程序才会向下执行。这两种区别的表现是：AlertDialog弹出时，背景是黑色的，但是当我们点击背景，AlertDialog会消失，证明程序不仅响应AlertDialog的操作，还响应其他操作，其他程序没有被阻塞，这说明了AlertDialog是非阻塞式对话框；PopupWindow弹出时，背景没有什么变化，但是当我们点击背景的时候，程序没有响应，只允许我们操作PopupWindow，其他操作被阻塞。

注意： 这里讲的阻塞并非线程阻塞，而是阻塞了其他UI操作，详情见：
PopupWindow的"阻塞"问题

一、前言

关于开源框架这块，其实主要是针对自己项目中使用到的框架进行准备。从使用，到使用场景、优缺点以及源码实现都需要逐一掌握理解。这一部分是向面试官展示自己水平与能力的一个重要部分，所以要着重准备。

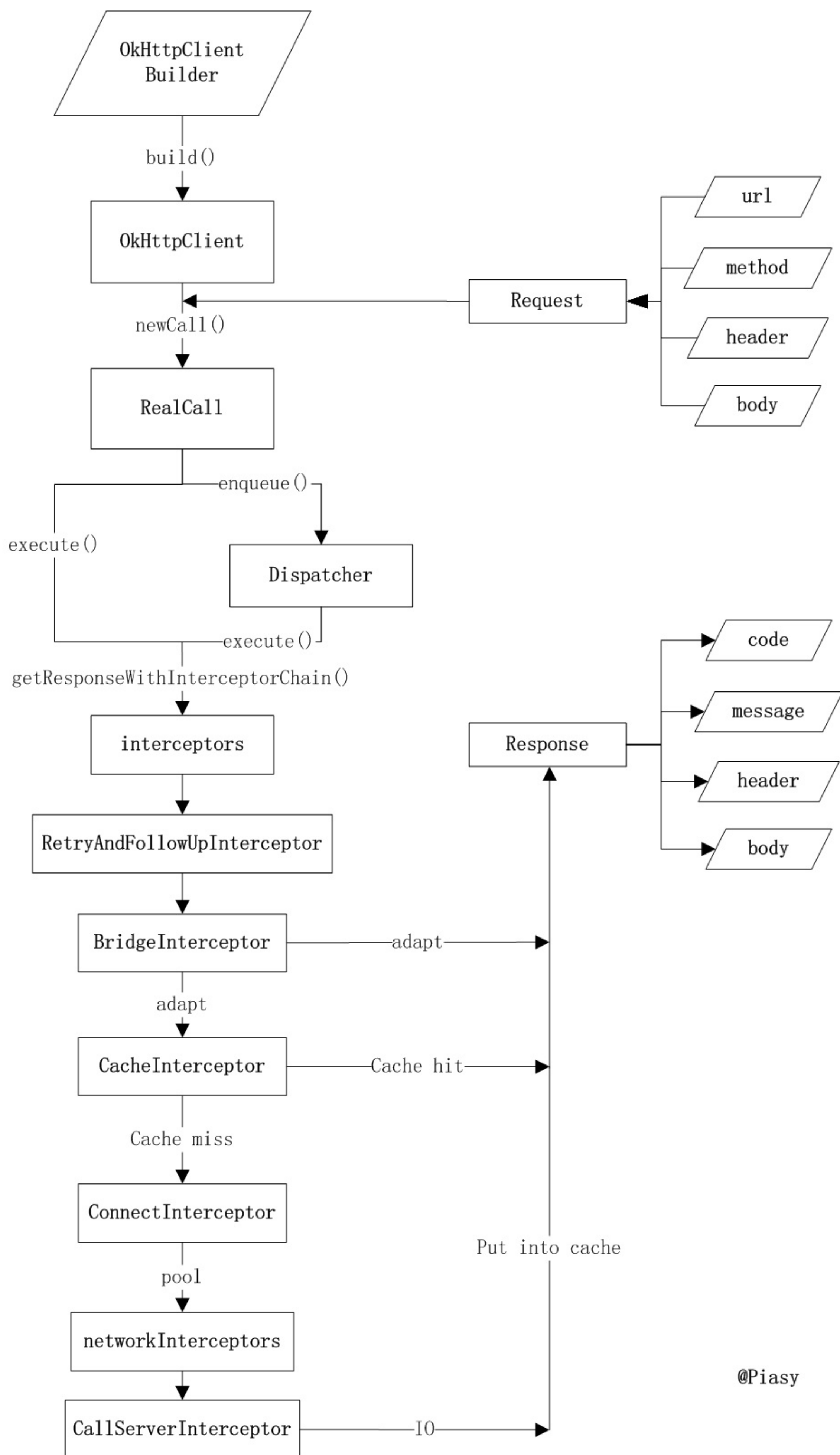
针对开源框架，该部分整理了自己在做项目过程中使用到的几个框架。大多都是从网上找的一些大佬的文章，深入浅出，着重讲述源码实现，可以帮助自己更好的理解。

二、目录

一、整体思路

从使用方法出发，首先是怎么使用，其次是我们使用的功能在内部是如何实现的，实现方案上有什么技巧，有什么范式。全文基本上是对 OkHttp 源码的一个分析与导读，非常建议大家下载 OkHttp 源码之后，跟着本文，过一遍源码。对于技巧和范式，由于目前我的功力还不到位，分析内容没多少，欢迎大家和我一起讨论。

首先放一张完整流程图（看不懂没关系，慢慢往后看）：


OkHttpClient
Builder
build()
OkHttpClient
Request
url
method
header
body
newCall()
RealCall
enqueue()
execute()
Dispatcher
execute()
getResponseWithInterceptorChain()
code
message
header
body
interceptors
Response
RetryAndFollowUpInterceptor
BridgeInterceptor
adapt
adapt
CacheInterceptor
Cache hit
Cache miss
ConnectInterceptor
Put into cache
pool
networkInterceptors
@Piasy
CallServerInterceptor
IO

二、基本用例

来自OkHttp 官方网站。

2.1.创建 OkHttpClient 对象

```
OkHttpClient client = new OkHttpClient();
```

咦，怎么不见 builder？莫急，且看其构造函数：

```
public OkHttpClient() {
  this(new Builder());
}
```

原来是方便我们使用，提供了一个“快捷操作”，全部使用了默认的配置。 `OkHttpClient.Builder` 类成员很多，后面我们再慢慢分析，这里先暂时略过：

```
public Builder() {
  dispatcher = new Dispatcher();
  protocols = DEFAULT_PROTOCOLS;
  connectionSpecs = DEFAULT_CONNECTION_SPECS;
  proxySelector = ProxySelector.getDefault();
  cookieJar = CookieJar.NO_COOKIES;
  socketFactory = SocketFactory.getDefault();
  hostnameVerifier = OkHostnameVerifier.INSTANCE;
  certificatePinner = CertificatePinner.DEFAULT;
  proxyAuthenticator = Authenticator.NONE;
  authenticator = Authenticator.NONE;
  connectionPool = new ConnectionPool();
  dns = Dns.SYSTEM;
  followSslRedirects = true;
  followRedirects = true;
  retryOnConnectionFailure = true;
  connectTimeout = 10_000;
  readTimeout = 10_000;
  writeTimeout = 10_000;
}
```

2.2.发起 HTTP 请求

```
String run(String url) throws IOException {
  Request request = new Request.Builder()
      .url(url)
      .build();

  Response response = client.newCall(request).execute();
  return response.body().string();
}
```

`OkHttpClient` 实现了 `Call.Factory`，负责根据请求创建新的 `Call`。

那我们现在就来看看它是如何创建 Call 的：

```
/**
  * Prepares the {@code request} to be executed at some point in
 the future.
  */
@Override public Call newCall(Request request) {
  return new RealCall(this, request);
}
```

如此看来功劳全在 `RealCall` 类了，下面我们一边分析同步网络请求的过程，一边了解 `RealCall` 的具体内容。

2.2.1.同步网络请求

我们首先看 `RealCall#execute` ：

```
@Override public Response execute() throws IOException {
  synchronized (this) {
    if (executed) throw new IllegalStateException("Already Execu
ted");  // (1)
    executed = true;
  }
  try {
    client.dispatcher().executed(this);
        // (2)
    Response result = getResponseWithInterceptorChain();
        // (3)
    if (result == null) throw new IOException("Canceled");
    return result;
  } finally {
    client.dispatcher().finished(this);
        // (4)
  }
}
```

这里我们做了 4 件事：

1. 检查这个 call 是否已经被执行了，每个 call 只能被执行一次，如果想要一个完全一样的 call，可以利用 `call#clone` 方法进行克隆。

2. 利用 `client.dispatcher().executed(this)` 来进行实际执行 `dispatcher` 是刚才看到的 `OkHttpClient.Builder` 的成员之一，它的文档说自己是异步 HTTP 请求的执行策略，现在看来，同步请求它也有掺和。
3. 调用 `getResponseWithInterceptorChain()` 函数获取 HTTP 返回结果，从函数名可以看出，这一步还会进行一系列“拦截”操作。
4. 最后还要通知 `dispatcher` 自己已经执行完毕。

dispatcher 这里我们不过度关注，在同步执行的流程中，涉及到 dispatcher 的内容只不过是告知它我们的执行状态，比如开始执行了（调用 `executed` ），比如执行完毕了（调用 `finished` ），在异步执行流程中它会有更多的参与。

真正发出网络请求，解析返回结果的，还是 `getResponseWithInterceptorChain` ：

```
private Response getResponseWithInterceptorChain() throws IOExce
ption {
  // Build a full stack of interceptors.
  List<Interceptor> interceptors = new ArrayList<>();
  interceptors.addAll(client.interceptors());
  interceptors.add(retryAndFollowUpInterceptor);
  interceptors.add(new BridgeInterceptor(client.cookieJar()));
  interceptors.add(new CacheInterceptor(client.internalCache()))
;
  interceptors.add(new ConnectInterceptor(client));
  if (!retryAndFollowUpInterceptor.isForWebSocket()) {
    interceptors.addAll(client.networkInterceptors());
  }
  interceptors.add(new CallServerInterceptor(
      retryAndFollowUpInterceptor.isForWebSocket()));

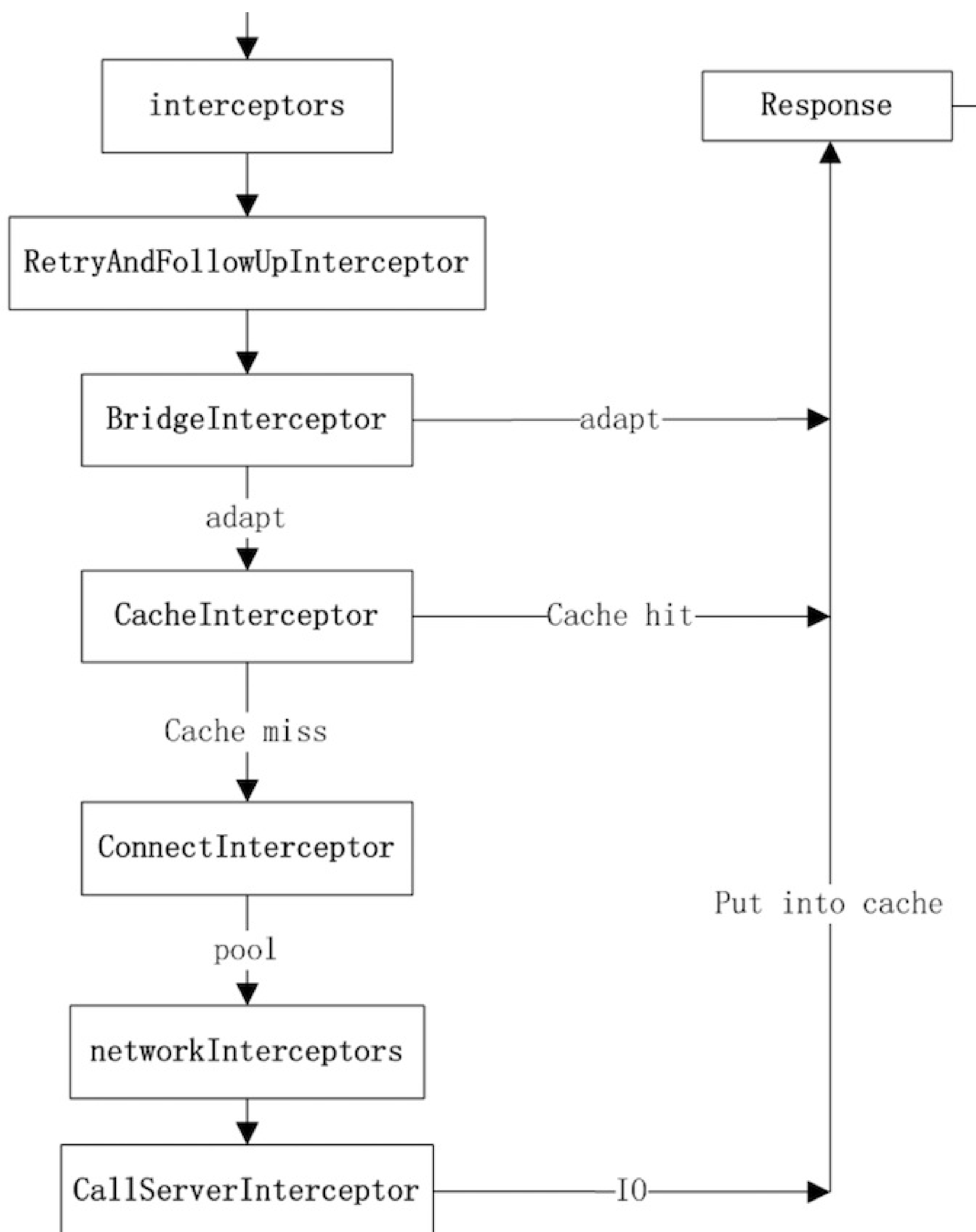
  Interceptor.Chain chain = new RealInterceptorChain(
      interceptors, null, null, null, 0, originalRequest);
  return chain.proceed(originalRequest);
}
```

在OkHttp 开发者之一介绍 OkHttp 的文章里面，作者讲到：

> the whole thing is just a stack of built-in interceptors.

可见 `Interceptor` 是 OkHttp 最核心的一个东西，不要误以为它只负责拦截请求进行一些额外的处理（例如 cookie），实际上它把实际的网络请求、缓存、透明压缩等功能都统一了起来，每一个功能都只是一个 `Interceptor` ，它们再连接成一个 `Interceptor.Chain` ，环环相扣，最终圆满完成一次网络请求。

从 `getResponseWithInterceptorChain` 函数我们可以看到 `Interceptor.Chain` 的分布依次是：



1. 在配置 `OkHttpClient` 时设置的 `interceptors` ；
2. 负责失败重试以及重定向的 `RetryAndFollowUpInterceptor` ；
3. 负责把用户构造的请求转换为发送到服务器的请求、把服务器返回的响应转换为用户友好的响应的 `BridgeInterceptor` ；
4. 负责读取缓存直接返回、更新缓存的 `CacheInterceptor` ；
5. 负责和服务器建立连接的 `ConnectInterceptor` ；
6. 配置 `OkHttpClient` 时设置的 `networkInterceptors` ；
7. 负责向服务器发送请求数据、从服务器读取响应数据 `CallServerInterceptor` 。

在这里，位置决定了功能，最后一个 Interceptor 一定是负责和服务器实际通讯的，重定向、缓存等一定是在实际通讯之前的。

责任链模式在这个 `Interceptor` 链条中得到了很好的实践。

> 它包含了一些命令对象和一系列的处理对象，每一个处理对象决定它能处理哪些命令对象，它也知道如何将它不能处理的命令对象传递给该链中的下一个处理对象。该模式还描述了往该处理链的末尾添加新的处理对象的方法。

对于把 `Request` 变成 `Response` 这件事来说，每个 `Interceptor` 都可能完成这件事，所以我们循着链条让每个 `Interceptor` 自行决定能否完成任务以及怎么完成任务（自力更生或者交给下一个 `Interceptor` ）。这样一来，完成网络请求这件事就彻底从 `RealCall` 类中剥离了出来，简化了各自的责任和逻辑。两个字：优雅！

责任链模式在安卓系统中也有比较典型的实践，例如 view 系统对点击事件（TouchEvent）的处理。

回到 OkHttp，在这里我们先简单分析一下 `ConnectInterceptor` 和 `CallServerInterceptor` ，看看 OkHttp 是怎么进行和服务器的实际通信的。

2.2.1.1.建立连接： `ConnectInterceptor`

```
@Override public Response intercept(Chain chain) throws IOException {
  RealInterceptorChain realChain = (RealInterceptorChain) chain;
  Request request = realChain.request();
  StreamAllocation streamAllocation = realChain.streamAllocation();

  // We need the network to satisfy this request. Possibly for validating a conditional GET.
  boolean doExtensiveHealthChecks = !request.method().equals("GET");
  HttpCodec httpCodec = streamAllocation.newStream(client, doExtensiveHealthChecks);
  RealConnection connection = streamAllocation.connection();

  return realChain.proceed(request, streamAllocation, httpCodec, connection);
}
```

实际上建立连接就是创建了一个 `HttpCodec` 对象，它将在后面的步骤中被使用，那它又是何方神圣呢？它是对 HTTP 协议操作的抽象，有两个实现： `Http1Codec` 和 `Http2Codec` ，顾名思义，它们分别对应 HTTP/1.1 和 HTTP/2 版本的实现。

在 `Http1Codec` 中，它利用Okio对 `Socket` 的读写操作进行封装，Okio 以后有机会再进行分析，现在让我们对它们保持一个简单地认识：它对 `java.io` 和 `java.nio` 进行了封装，让我们更便捷高效的进行 IO 操作。

而创建 `HttpCodec` 对象的过程涉及到 `StreamAllocation` 、 `RealConnection` ，代码较长，这里就不展开，这个过程概括来说，就是找到一个可用的 `RealConnection` ，再利用 `RealConnection` 的输入输出（ `BufferedSource` 和 `BufferedSink` ）创建 `HttpCodec` 对象，供后续步骤使用。

2.2.1.2.发送和接收数据： `CallServerInterceptor`

```
@Override public Response intercept(Chain chain) throws IOException {
```

```
  HttpCodec httpCodec = ((RealInterceptorChain) chain).httpStrea
m();
  StreamAllocation streamAllocation = ((RealInterceptorChain) ch
ain).streamAllocation();
  Request request = chain.request();

  long sentRequestMillis = System.currentTimeMillis();
  httpCodec.writeRequestHeaders(request);

  if (HttpMethod.permitsRequestBody(request.method()) && request
.body() != null) {
    Sink requestBodyOut = httpCodec.createRequestBody(request, r
equest.body().contentLength());
    BufferedSink bufferedRequestBody = Okio.buffer(requestBodyOu
t);
    request.body().writeTo(bufferedRequestBody);
    bufferedRequestBody.close();
  }

  httpCodec.finishRequest();

  Response response = httpCodec.readResponseHeaders()
      .request(request)
      .handshake(streamAllocation.connection().handshake())
      .sentRequestAtMillis(sentRequestMillis)
      .receivedResponseAtMillis(System.currentTimeMillis())
      .build();

  if (!forWebSocket || response.code() != 101) {
    response = response.newBuilder()
        .body(httpCodec.openResponseBody(response))
        .build();
  }

  if ("close".equalsIgnoreCase(response.request().header("Connec
tion"))
      || "close".equalsIgnoreCase(response.header("Connection"))
) {
    streamAllocation.noNewStreams();
  }
```

```
  // 省略部分检查代码

  return response;
}
```

我们抓住主干部分：

1. 向服务器发送 request header；
2. 如果有 request body，就向服务器发送；
3. 读取 response header，先构造一个 `Response` 对象；
4. 如果有 response body，就在 3 的基础上加上 body 构造一个新的 `Response` 对象；

这里我们可以看到，核心工作都由 `HttpCodec` 对象完成，而 `HttpCodec` 实际上利用的是 Okio，而 Okio 实际上还是用的 `Socket` ，所以没什么神秘的，只不过一层套一层，层数有点多。

其实 `Interceptor` 的设计也是一种分层的思想，每个 `Interceptor` 就是一层。为什么要套这么多层呢？分层的思想在 TCP/IP 协议中就体现得淋漓尽致，分层简化了每一层的逻辑，每层只需要关注自己的责任（单一原则思想也在此体现），而各层之间通过约定的接口/协议进行合作（面向接口编程思想），共同完成复杂的任务。

简单应该是我们的终极追求之一，尽管有时为了达成目标不得不复杂，但如果有另一种更简单的方式，我想应该没有人不愿意替换。

2.2.2.发起异步网络请求

```
client.newCall(request).enqueue(new Callback() {
    @Override
    public void onFailure(Call call, IOException e) {
    }

    @Override
    public void onResponse(Call call, Response response) throws
IOException {
        System.out.println(response.body().string());
    }
});

// RealCall#enqueue
@Override public void enqueue(Callback responseCallback) {
  synchronized (this) {
    if (executed) throw new IllegalStateException("Already Execu
ted");
    executed = true;
  }
  client.dispatcher().enqueue(new AsyncCall(responseCallback));
}

// Dispatcher#enqueue
synchronized void enqueue(AsyncCall call) {
  if (runningAsyncCalls.size() < maxRequests && runningCallsForH
ost(call) < maxRequestsPerHost) {
    runningAsyncCalls.add(call);
    executorService().execute(call);
  } else {
    readyAsyncCalls.add(call);
  }
}
```

这里我们就能看到 dispatcher 在异步执行时发挥的作用了，如果当前还能执行一个并发请求，那就立即执行，否则加入 `readyAsyncCalls` 队列，而正在执行的请求执行完毕之后，会调用 `promoteCalls()` 函数，来把 `readyAsyncCalls` 队列中的 `AsyncCall` “提升”为 `runningAsyncCalls` ，并开始执行。

这里的 `AsyncCall` 是 `RealCall` 的一个内部类，它实现了 `Runnable` ，所以可以被提交到 `ExecutorService` 上执行，而它在执行时会调用 `getResponseWithInterceptorChain()` 函数，并把结果通过 `responseCallback` 传递给上层使用者。

这样看来，同步请求和异步请求的原理是一样的，都是在 `getResponseWithInterceptorChain()` 函数中通过 `Interceptor` 链条来实现的网络请求逻辑，而异步则是通过 `ExecutorService` 实现。

2.3返回数据的获取

在上述同步（ `Call#execute()` 执行之后）或者异步（ `Callback#onResponse()` 回调中）请求完成之后，我们就可以从 `Response` 对象中获取到响应数据了，包括 HTTP status code，status message，response header，response body 等。这里 body 部分最为特殊，因为服务器返回的数据可能非常大，所以必须通过数据流的方式来进行访问（当然也提供了诸如 `string()` 和 `bytes()` 这样的方法将流内的数据一次性读取完毕），而响应中其他部分则可以随意获取。

响应 body 被封装到 `ResponseBody` 类中，该类主要有两点需要注意：

1. 每个 body 只能被消费一次，多次消费会抛出异常；
2. body 必须被关闭，否则会发生资源泄漏；

在2.2.1.2.发送和接收数据：CallServerInterceptor小节中，我们就看过了 body 相关的代码：

```
if (!forWebSocket || response.code() != 101) {
  response = response.newBuilder()
      .body(httpCodec.openResponseBody(response))
      .build();
}
```

由 `HttpCodec#openResponseBody` 提供具体 HTTP 协议版本的响应 body，而 `HttpCodec` 则是利用 Okio 实现具体的数据 IO 操作。

这里有一点值得一提，OkHttp 对响应的校验非常严格，HTTP status line 不能有任何杂乱的数据，否则就会抛出异常，在我们公司项目的实践中，由于服务器的问题，偶尔 status line 会有额外数据，而服务端的问题也毫无头绪，导致我们不得不忍痛继续使用 HttpUrlConnection，而后者在一些系统上又存在各种其他的问题，例如魅族系统发送 multi-part form 的时候就会出现没有响应的问题。

2.4.HTTP 缓存

在2.2.1.同步网络请求小节中，我们已经看到了 `Interceptor` 的布局，在建立连接、和服务器通讯之前，就是 `CacheInterceptor` ，在建立连接之前，我们检查响应是否已经被缓存、缓存是否可用，如果是则直接返回缓存的数据，否则就进行后面的流程，并在返回之前，把网络的数据写入缓存。

这块代码比较多，但也很直观，主要涉及 HTTP 协议缓存细节的实现，而具体的缓存逻辑 OkHttp 内置封装了一个 `Cache` 类，它利用 `DiskLruCache` ，用磁盘上的有限大小空间进行缓存，按照 LRU 算法进行缓存淘汰，这里也不再展开。

我们可以在构造 `OkHttpClient` 时设置 `Cache` 对象，在其构造函数中我们可以指定目录和缓存大小：

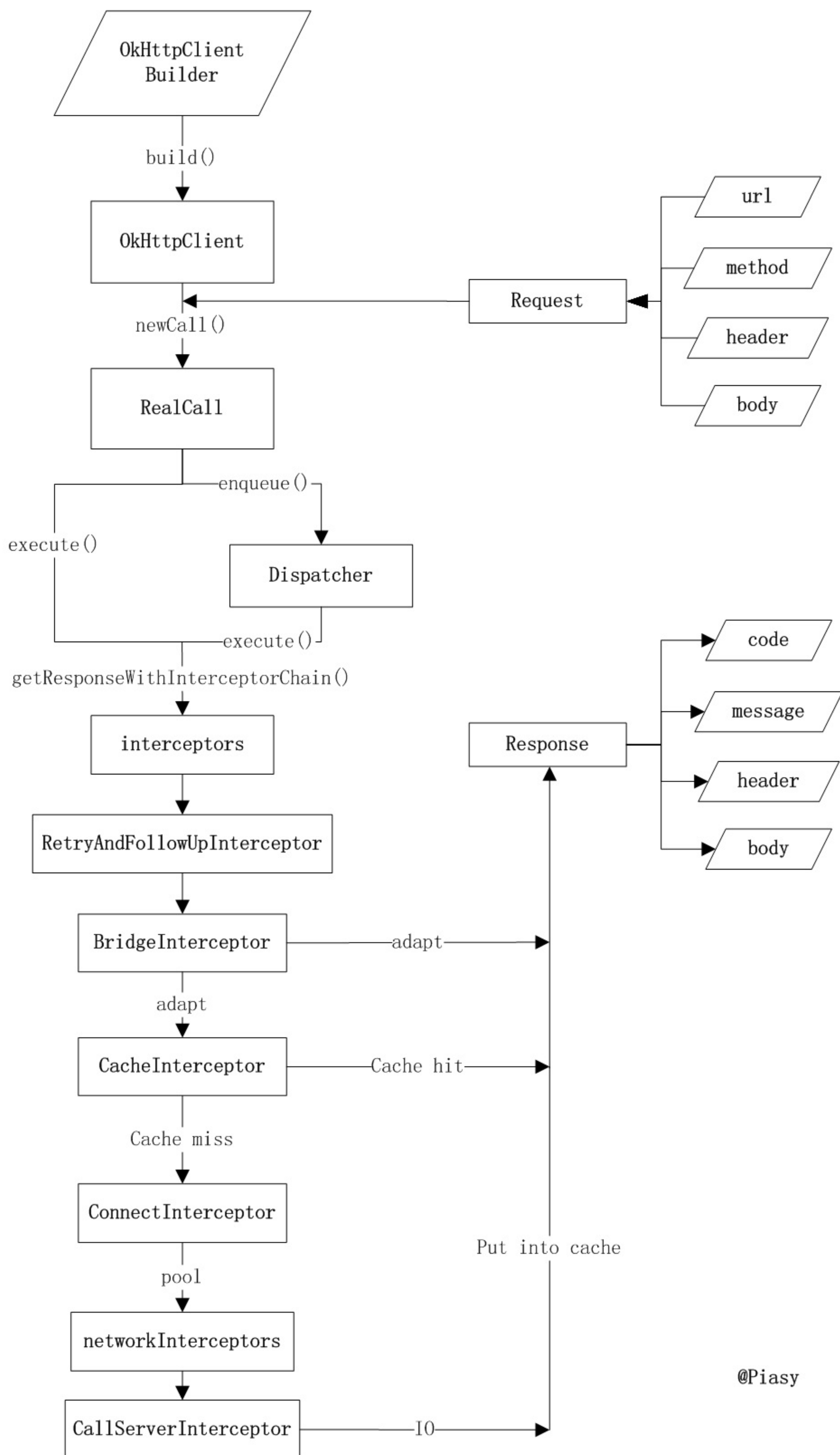
```
public Cache(File directory, long maxSize);
```

而如果我们对 OkHttp 内置的 `Cache` 类不满意，我们可以自行实现 `InternalCache` 接口，在构造 `OkHttpClient` 时进行设置，这样就可以使用我们自定义的缓存策略了。

三、总结

OkHttp 还有很多细节部分没有在本文展开，例如 HTTP2/HTTPS 的支持等，但建立一个清晰的概览非常重要。对整体有了清晰认识之后，细节部分如有需要，再单独深入将更加容易。

在文章最后我们再来回顾一下完整的流程图：



OkHttpClient
Builder
build()
OkHttpClient
Request
url
method
header
body
newCall()
RealCall
enqueue()
execute()
Dispatcher
execute()
getResponseWithInterceptorChain()
interceptors
Response
code
message
header
body
RetryAndFollowUpInterceptor
BridgeInterceptor
adapt
adapt
CacheInterceptor
Cache hit
Cache miss
ConnectInterceptor
Put into cache
pool
networkInterceptors
@Piasy
CallServerInterceptor
IO

- `OkHttpClient` 实现 `Call.Factory` ，负责为 `Request` 创建 `Call` ；
- `RealCall` 为具体的 `Call` 实现，其 `enqueue()` 异步接口通过 `Dispatcher` 利用 `ExecutorService` 实现，而最终进行网络请求时和同步 `execute()` 接口一致，都是通过 `getResponseWithInterceptorChain()` 函数实现；
- `getResponseWithInterceptorChain()` 中利用 `Interceptor` 链条，分层实现缓存、透明压缩、网络 IO 等功能；

一、整体思路

从使用方法出发，首先是怎么使用，其次是我们使用的功能在内部是如何实现的，实现方案上有什么技巧，有什么范式。全文基本上是对 Retrofit 源码的一个分析与导读，非常建议大家下载 Retrofit 源码之后，跟着本文，过一遍源码。

二、基本用例

2.1 创建 Retrofit 对象

```
Retrofit retrofit = new Retrofit.Builder()
    .baseUrl("https://api.github.com/")
    .addConverterFactory(GsonConverterFactory.create())
    .build();
```

builder 模式，外观模式（门面模式），这就不多说了，可以看看 stay 的 Retrofit 分析-经典设计模式案例这篇文章。

2.2 定义 API 并获取 API 实例

```
public interface GitHubService {
  @GET("users/{user}/repos")
  Call<List<Repo>> listRepos(@Path("user") String user);
}

GitHubService github = retrofit.create(GitHubService.class);
```

先看定义，非常简洁，也没有什么特别之处，除了两个注解： `@GET` 和 `@Path` 。它们的用处稍后再分析，我们接着看创建 API 实例： `retrofit.create(GitHubService.class)` 。这样就创建了 API 实例了，就可以调用 API 的方法发起 HTTP 网络请求了，太方便了。

但 `create` 方法是怎么创建 API 实例的呢？

```
public <T> T create(final Class<T> service) {
  // 省略非关键代码
  return (T) Proxy.newProxyInstance(service.getClassLoader(),
      new Class<?>[] { service },
      new InvocationHandler() {
        @Override
        public Object invoke(Object proxy, Method method, Object
... args)
            throws Throwable {
          // 先省略实现
        }
      });
}
```

创建 API 实例使用的是动态代理技术。

简而言之，就是动态生成接口的实现类（当然生成实现类有缓存机制），并创建其实例（称之为代理），代理把对接口的调用转发给 `InvocationHandler` 实例，而在 `InvocationHandler` 的实现中，除了执行真正的逻辑（例如再次转发给真正的实现类对象），我们还可以进行一些有用的操作，例如统计执行时间、进行初始化和清理、对接口调用进行检查等。

为什么要用动态代理？因为对接口的所有方法的调用都会集中转发到 `InvocationHandler#invoke` 函数中，我们可以集中进行处理，更方便了。你可能会想，我也可以手写这样的代理类，把所有接口的调用都转发到 `InvocationHandler#invoke` 呀，当然可以，但是可靠地自动生成岂不更方便？

2.3 调用 API 方法

获取到 API 实例之后，调用方法和普通的代码没有任何区别：

```
Call<List<Repo>> call = github.listRepos("square");
List<Repo> repos = call.execute().body();
```

这两行代码就发出了 HTTP 请求，并把返回的数据转化为了 `List<Repo>` ，太方便了！

现在我们来看看调用 `listRepos` 是怎么发出 HTTP 请求的。上面 `Retrofit#create` 方法返回时省略的代码如下：

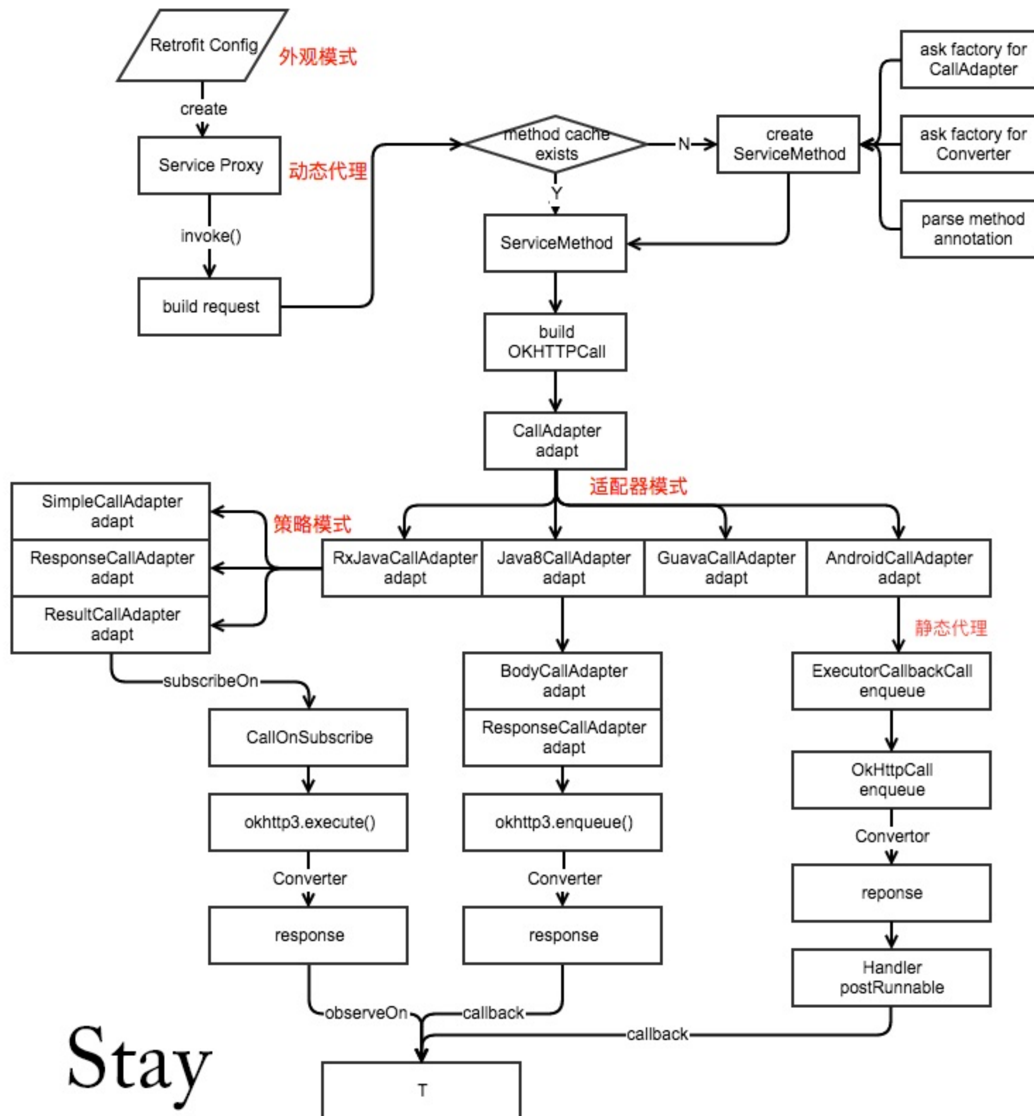
```
return (T) Proxy.newProxyInstance(service.getClassLoader(),
    new Class<?>[] { service },
    new InvocationHandler() {
      private final Platform platform = Platform.get();

      @Override
      public Object invoke(Object proxy, Method method, Object..
. args)
          throws Throwable {
        // If the method is a method from Object then defer to n
ormal invocation.
        if (method.getDeclaringClass() == Object.class) {
          return method.invoke(this, args);
        }
        if (platform.isDefaultMethod(method)) {
          return platform.invokeDefaultMethod(method, service, p
roxy, args);
        }
        ServiceMethod serviceMethod = loadServiceMethod(method);
        OkHttpCall okHttpCall = new OkHttpCall<>(serviceMethod,
args);
        return serviceMethod.callAdapter.adapt(okHttpCall);
      }
    });
```

如果调用的是 `Object` 的方法，例如 `equals` ， `toString` ，那就直接调用。如果是 default 方法（Java 8 引入），就调用 default 方法。这些我们都先不管，因为我们在安卓平台调用 `listRepos` ，肯定不是这两种情况，那这次调用真正干活的就是这三行代码了（好好记住这三行代码，因为接下来很长的篇幅都是在讲它们 :) ）：

```
ServiceMethod serviceMethod = loadServiceMethod(method);
OkHttpCall okHttpCall = new OkHttpCall<>(serviceMethod, args);
return serviceMethod.callAdapter.adapt(okHttpCall);
```

在继续分析这三行代码之前，我们先看看 Stay 在 Retrofit分析-漂亮的解耦套路 这篇文章中分享的流程图，完整的流程概览建议仔细看看这篇文章：



这三行代码基本就是对应于流程图中轴上部了，`ServiceMethod`，`build OkHttpCall`，`CallAdapter adapt`。

2.4 ServiceMethod

`ServiceMethod<T>` 类的作用正如其 JavaDoc 所言：

> Adapts an invocation of an interface method into an HTTP call. 把对接口方法的调用转为一次 HTTP 调用。

一个 `ServiceMethod` 对象对应于一个 API interface 的一个方法，`loadServiceMethod(method)` 方法负责加载 `ServiceMethod`：

```
ServiceMethod loadServiceMethod(Method method) {
  ServiceMethod result;
  synchronized (serviceMethodCache) {
    result = serviceMethodCache.get(method);
    if (result == null) {
      result = new ServiceMethod.Builder(this, method).build();
      serviceMethodCache.put(method, result);
    }
  }
  return result;
}
```

这里实现了缓存逻辑，同一个 API 的同一个方法，只会创建一次。这里由于我们每次获取 API 实例都是传入的 `class` 对象，而 `class` 对象是进程内单例的，所以获取到它的同一个方法 `Method` 实例也是单例的，所以这里的缓存是有效的。

我们再看看 `ServiceMethod` 的构造函数：

```
ServiceMethod(Builder<T> builder) {
  this.callFactory = builder.retrofit.callFactory();
  this.callAdapter = builder.callAdapter;
  this.baseUrl = builder.retrofit.baseUrl();
  this.responseConverter = builder.responseConverter;
  this.httpMethod = builder.httpMethod;
  this.relativeUrl = builder.relativeUrl;
  this.headers = builder.headers;
  this.contentType = builder.contentType;
  this.hasBody = builder.hasBody;
  this.isFormEncoded = builder.isFormEncoded;
  this.isMultipart = builder.isMultipart;
  this.parameterHandlers = builder.parameterHandlers;
}
```

成员很多，但这里我们重点关注四个成员：`callFactory`，`callAdapter`，`responseConverter` 和 `parameterHandlers`。

1. `callFactory` 负责创建 HTTP 请求，HTTP 请求被抽象为了

`okhttp3.Call` 类，它表示一个已经准备好，可以随时执行的 HTTP 请求；
2. `callAdapter` 把 `retrofit2.Call<T>` 转为 `T` （注意和 `okhttp3.Call` 区分开来， `retrofit2.Call<T>` 表示的是对一个 Retrofit 方法的调用），这个过程会发送一个 HTTP 请求，拿到服务器返回的数据（通过 `okhttp3.Call` 实现），并把数据转换为声明的 `T` 类型对象（通过 `Converter<F, T>` 实现）；
3. `responseConverter` 是 `Converter<ResponseBody, T>` 类型，负责把服务器返回的数据（JSON、XML、二进制或者其他格式，由 `ResponseBody` 封装）转化为 `T` 类型的对象；
4. `parameterHandlers` 则负责解析 API 定义时每个方法的参数，并在构造 HTTP 请求时设置参数；

它们的使用稍后再分析，这里先看看它们的创建（代码比较分散，就不贴太多代码了，大多是结论）：

2.4.1 callFactory

`this.callFactory = builder.retrofit.callFactory()` ，所以 `callFactory` 实际上由 `Retrofit` 类提供，而我们在构造 `Retrofit` 对象时，可以指定 `callFactory` ，如果不指定，将默认设置为一个 `okhttp3.OkHttpClient` 。

2.4.2 callAdapter

```
private CallAdapter<?> createCallAdapter() {
  // 省略检查性代码
  Annotation[] annotations = method.getAnnotations();
  try {
    return retrofit.callAdapter(returnType, annotations);
  } catch (RuntimeException e) {
    // Wide exception range because factories are user code.
    throw methodError(e, "Unable to create call adapter for %s",
 returnType);
  }
}
```

可以看到，`callAdapter` 还是由 `Retrofit` 类提供。在 `Retrofit` 类内部，将遍历一个 `CallAdapter.Factory` 列表，让工厂们提供，如果最终没有工厂能（根据 `returnType` 和 `annotations` ）提供需要的 `CallAdapter` ，那将抛出异常。而这个工厂列表我们可以在构造 `Retrofit` 对象时进行添加。

2.4.3、 `responseConverter`

```
private Converter<ResponseBody, T> createResponseConverter() {
  Annotation[] annotations = method.getAnnotations();
  try {
    return retrofit.responseBodyConverter(responseType, annotati
ons);
  } catch (RuntimeException e) {
    // Wide exception range because factories are user code.
    throw methodError(e, "Unable to create converter for %s", re
sponseType);
  }
}
```

同样， `responseConverter` 还是由 `Retrofit` 类提供，而在其内部，逻辑和创建 `callAdapter` 基本一致，通过遍历 `Converter.Factory` 列表，看看有没有工厂能够提供需要的 responseBodyConverter。工厂列表同样可以在构造 `Retrofit` 对象时进行添加。

2.4.4 parameterHandlers

每个参数都会有一个 `ParameterHandler` ，由 `ServiceMethod#parseParameter` 方法负责创建，其主要内容就是解析每个参数使用的注解类型（诸如 `Path` ， `Query` ， `Field` 等），对每种类型进行单独的处理。构造 HTTP 请求时，我们传递的参数都是字符串，那 Retrofit 是如何把我们传递的各种参数都转化为 String 的呢？还是由 `Retrofit` 类提供 converter！

`Converter.Factory` 除了提供上一小节提到的 responseBodyConverter，还提供 requestBodyConverter 和 stringConverter，API 方法中除了 `@Body` 和 `@Part` 类型的参数，都利用 stringConverter 进行转换，而 `@Body` 和 `@Part` 类型的参数则利用 requestBodyConverter 进行转换。

这三种 converter 都是通过“询问”工厂列表进行提供，而工厂列表我们可以在构造 `Retrofit` 对象时进行添加。

2.4.5 工厂让各个模块得以高度解耦

上面提到了三种工厂：`okhttp3.Call.Factory`，`CallAdapter.Factory` 和 `Converter.Factory`，分别负责提供不同的模块，至于怎么提供、提供何种模块，统统交给工厂，Retrofit 完全不掺和，它只负责提供用于决策的信息，例如参数/返回值类型、注解等。

这不正是我们苦苦追求的高内聚低耦合效果吗？解耦的第一步就是面向接口编程，模块之间、类之间通过接口进行依赖，创建怎样的实例，则交给工厂负责，工厂同样也是接口，添加（Retrofit doc 中使用 install 安装一词，非常贴切）怎样的工厂，则在最初构造 `Retrofit` 对象时决定，各个模块之间完全解耦，每个模块只专注于自己的职责，全都是套路，值得反复玩味、学习与模仿。

除了上面重点分析的这四个成员，`ServiceMethod` 中还包含了 API 方法的 url 解析等逻辑，包含了众多关于泛型和反射相关的代码，有类似需求的时候，也非常值得学习模仿。

2.5 OkHttpCall

终于把 `ServiceMethod` 看了个大概，接下来我们看看 `OkHttpCall`。

`OkHttpCall` 实现了 `retrofit2.Call`，我们通常会使用它的 `execute()` 和 `enqueue(Callback<T> callback)` 接口。前者用于同步执行 HTTP 请求，后者用于异步执行。

2.5.1，先看 `execute()`

```
@Override
public Response<T> execute() throws IOException {
  okhttp3.Call call;

  synchronized (this) {
    // 省略部分检查代码

    call = rawCall;
```

```
    if (call == null) {
      try {
        call = rawCall = createRawCall();
      } catch (IOException | RuntimeException e) {
        creationFailure = e;
        throw e;
      }
    }
  }

  return parseResponse(call.execute());
}

private okhttp3.Call createRawCall() throws IOException {
  Request request = serviceMethod.toRequest(args);
  okhttp3.Call call = serviceMethod.callFactory.newCall(request)
;
  if (call == null) {
    throw new NullPointerException("Call.Factory returned null."
);
  }
  return call;
}

Response<T> parseResponse(okhttp3.Response rawResponse) throws I
OException {
  ResponseBody rawBody = rawResponse.body();

  // Remove the body's source (the only stateful object) so we c
an pass the response along.
  rawResponse = rawResponse.newBuilder()
      .body(new NoContentResponseBody(rawBody.contentType(), raw
Body.contentLength()))
      .build();

  int code = rawResponse.code();
  if (code < 200 || code >= 300) {
    // ...返回错误
  }
```

```
    if (code == 204 || code == 205) {
      return Response.success(null, rawResponse);
    }

    ExceptionCatchingRequestBody catchingBody = new ExceptionCatch
  ingRequestBody(rawBody);
    try {
      T body = serviceMethod.toResponse(catchingBody);
      return Response.success(body, rawResponse);
    } catch (RuntimeException e) {
      // ...异常处理
    }
  }
```

主要包括三步：

1. 创建 `okhttp3.Call` ，包括构造参数；
2. 执行网络请求；
3. 解析网络请求返回的数据；

`createRawCall()` 函数中，我们调用了 `serviceMethod.toRequest(args)` 来创建 `okhttp3.Request` ，而在后者中，我们之前准备好的 `parameterHandlers` 就派上了用场。

然后我们再调用 `serviceMethod.callFactory.newCall(request)` 来创建 `okhttp3.Call` ，这里之前准备好的 `callFactory` 同样也派上了用场，由于工厂在构造 `Retrofit` 对象时可以指定，所以我们也可以指定其他的工厂（例如使用过时的 `HttpURLConnection` 的工厂），来使用其它的底层 HttpClient 实现。

我们调用 `okhttp3.Call#execute()` 来执行网络请求，这个方法是阻塞的，执行完毕之后将返回收到的响应数据。收到响应数据之后，我们进行了状态码的检查，通过检查之后我们调用了 `serviceMethod.toResponse(catchingBody)` 来把响应数据转化为了我们需要的数据类型对象。在 `toResponse` 函数中，我们之前准备好的 `responseConverter` 也派上了用场。

好了，之前准备好的东西都派上了用场，还好没有白费 :)

2.5.2 再看 `enqueue(Callback<T> callback)`

这里的异步交给了 `okhttp3.Call#enqueue(Callback responseCallback)` 来实现，并在它的 callback 中调用 `parseResponse` 解析响应数据，并转发给传入的 callback。

2.6 CallAdapter

终于到了最后一步了， `CallAdapter<T>#adapt(Call<R> call)` 函数负责把 `retrofit2.Call<R>` 转为 `T` 。这里 `T` 当然可以就是 `retrofit2.Call<R>` ，这时我们直接返回参数就可以了，实际上这正是 `DefaultCallAdapterFactory` 创建的 `CallAdapter` 的行为。至于其他类型的工厂返回的 `CallAdapter` 的行为，这里暂且不表，后面再单独分析。

至此，一次对 API 方法的调用是如何构造并发起网络请求、以及解析返回数据，这整个过程大致是分析完毕了。对整个流程的概览非常重要，结合 stay 画的流程图，应该能够比较轻松地看清整个流程了。

虽然我们还没分析完，不过也相当于到了万里长征的遵义，终于可以舒一口气了 :)

三、retrofit-adapters 模块

retrofit 模块内置了 `DefaultCallAdapterFactory` 和 `ExecutorCallAdapterFactory` ，它们都适用于 API 方法得到的类型为 `retrofit2.Call` 的情形，前者生产的 adapter 啥也不做，直接把参数返回，后者生产的 adapter 则会在异步调用时在指定的 `Executor` 上执行回调。

retrofit-adapters 的各个子模块则实现了更多的工厂： `GuavaCallAdapterFactory` ， `Java8CallAdapterFactory` 和 `RxJavaCallAdapterFactory` 。这里我主要分析 `RxJavaCallAdapterFactory` ，下面的内容就需要一些 RxJava 的知识了，不过我想使用 Retrofit 的你，肯定也在使用 RxJava :)

`RxJavaCallAdapterFactory#get` 方法中对返回值的类型进行了检查，只支持 `rx.Single` ， `rx.Completable` 和 `rx.Observable` ，这里我主要关注对 `rx.Observable` 的支持。

`RxJavaCallAdapterFactory#getCallAdapter` 方法中对返回值的泛型类型进行了进一步检查，例如我们声明的返回值类型为 `Observable<List<Repo>>` ，泛型类型就是 `List<Repo>` ，这里对 `retrofit2.Response` 和

`retrofit2.adapter.rxjava.Result` 进行了特殊处理，有单独的 adapter 负责进行转换，其他所有类型都由 `SimpleCallAdapter` 负责转换。

那我们就来看看 `SimpleCallAdapter#adapt` ：

```
@Override
public <R> Observable<R> adapt(Call<R> call) {
  Observable<R> observable = Observable.create(new CallOnSubscri
be<>(call))
      .lift(OperatorMapResponseToBodyOrError.<R>instance());
  if (scheduler != null) {
    return observable.subscribeOn(scheduler);
  }
  return observable;
}
```

这里创建了一个 `Observable` ，它的逻辑由 `CallOnSubscribe` 类实现，同时使用了一个 `OperatorMapResponseToBodyOrError` 操作符，用来把 `retrofit2.Response` 转为我们声明的类型，或者错误异常类型。

我们接着看 `CallOnSubscribe#call` ：

```
@Override
public void call(final Subscriber<? super Response<T>> subscribe
r) {
  // Since Call is a one-shot type, clone it for each new subscr
iber.
  Call<T> call = originalCall.clone();

  // Wrap the call in a helper which handles both unsubscription
 and backpressure.
  RequestArbiter<T> requestArbiter = new RequestArbiter<>(call,
subscriber);
  subscriber.add(requestArbiter);
  subscriber.setProducer(requestArbiter);
}
```

代码很简短，只干了三件事：

1. clone 了原来的 call，因为 `okhttp3.Call` 是只能用一次的，所以每次都是新 clone 一个进行网络请求；
2. 创建了一个叫做 `RequestArbiter` 的 producer，别被它的名字吓懵了，它就是个 producer；
3. 把这个 producer 设置给 subscriber；

简言之，大部分情况下 Subscriber 都是被动接受 Observable push 过来的数据，但要是 Observable 发得太快，Subscriber 处理不过来，那就有问题了，所以就有了一种 Subscriber 主动 pull 的机制，而这种机制就是通过 Producer 实现的。给 Subscriber 设置 Producer 之后（通过 `Subscriber#setProducer` 方法），Subscriber 就会通过 Producer 向上游根据自己的能力请求数据（通过 `Producer#request` 方法），而 Producer 收到请求之后（通常都是 Observable 管理 Producer，所以"相当于"就是 Observable 收到了请求），再根据请求的量给 Subscriber 发数据。

那我们就看看 `RequestArbiter#request` ：

```
@Override
public void request(long n) {
  if (n < 0) throw new IllegalArgumentException("n < 0: " + n);
  if (n == 0) return; // Nothing to do when requesting 0.
  if (!compareAndSet(false, true)) return; // Request was alread
y triggered.

  try {
    Response<T> response = call.execute();
    if (!subscriber.isUnsubscribed()) {
      subscriber.onNext(response);
    }
  } catch (Throwable t) {
    Exceptions.throwIfFatal(t);
    if (!subscriber.isUnsubscribed()) {
      subscriber.onError(t);
    }
    return;
  }

  if (!subscriber.isUnsubscribed()) {
    subscriber.onCompleted();
  }
}
```

producer相关的逻辑非常简单，看看Operator 并发原语： producers（二），SingleDelayedProducer就能懂了，这里就不在赘述。实际干活的逻辑就是执行 `call.execute()` ，并把返回值发送给下游。

而 `OperatorMapResponseToBodyOrError#call` 也相当简短：

```
@Override
public Subscriber<? super Response<T>> call(final Subscriber<? s
uper T> child) {
  return new Subscriber<Response<T>>(child) {
    @Override
    public void onNext(Response<T> response) {
      if (response.isSuccessful()) {
        child.onNext(response.body());
      } else {
        child.onError(new HttpException(response));
      }
    }

    @Override
    public void onCompleted() {
      child.onCompleted();
    }

    @Override
    public void onError(Throwable e) {
      child.onError(e);
    }
  };
}
```

关键就是调用了 `response.body()` 并发送给下游。这里，`body()` 返回的就是我们声明的泛型类型了，至于 Retrofit 是怎么把服务器返回的数据转为我们声明的类型的，这就是 `responseConverter` 的事了，还记得吗？

最后看一张返回 `Observable` 时的调用栈：

```
"main"@1 in group "main": RUNNING
call:64, SimpleService$1 (com.example.retrofit)
call:62, SimpleService$1 (com.example.retrofit)
onNext:39, ActionSubscriber (rx.internal.util)
onNext:139, SafeSubscriber (rx.observers)
onNext:41, OperatorMapResponseToBodyOrError$1 (retrofit2.adapter.rxjava)
onNext:38, OperatorMapResponseToBodyOrError$1 (retrofit2.adapter.rxjava)
request:173, RxJavaCallAdapterFactory$RequestArbiter (retrofit2.adapter.rxjava)
setProducer:209, Subscriber (rx)
setProducer:205, Subscriber (rx)
setProducer:205, Subscriber (rx)
call:152, RxJavaCallAdapterFactory$CallOnSubscribe (retrofit2.adapter.rxjava)
call:138, RxJavaCallAdapterFactory$CallOnSubscribe (retrofit2.adapter.rxjava)
call:50, OnSubscribeLift (rx.internal.operators)
call:30, OnSubscribeLift (rx.internal.operators)
subscribe:8553, Observable (rx)
subscribe:8520, Observable (rx)
subscribe:8343, Observable (rx)
main:62, SimpleService (com.example.retrofit)
```

执行路径就是：

1. `Observable.subscribe` ，触发 API 调用的执行；
2. `CallOnSubscribe#call` ，clone call，创建并设置 producer；
3. `RequestArbiter#request` ，subscriber 被设置了 producer 之后最终调用 request，在 request 中发起请求，把结果发给下游；
4. `OperatorMapResponseToBodyOrError$1#onNext` ，把 response 的 body 发给下游；
5. 最终就到了我们 subscribe 时传入的回调里面了；

四、retrofit-converters 模块

retrofit 模块内置了 `BuiltInConverters` ，只能处理 `ResponseBody` ， `RequestBody` 和 `String` 类型的转化（实际上不需要转）。而 retrofit-converters 中的子模块则提供了 JSON，XML，ProtoBuf 等类型数据的转换功能，而且还有多种转换方式可以选择。这里我主要关注 `GsonConverterFactory` 。

代码非常简单：

```
@Override
public Converter<ResponseBody, ?> responseBodyConverter(Type typ
e,
    Annotation[] annotations, Retrofit retrofit) {
  TypeAdapter<?> adapter = gson.getAdapter(TypeToken.get(type));
  return new GsonResponseBodyConverter<>(gson, adapter);
}

final class GsonResponseBodyConverter<T> implements Converter<Re
sponseBody, T> {
  private final Gson gson;
  private final TypeAdapter<T> adapter;

  GsonResponseBodyConverter(Gson gson, TypeAdapter<T> adapter) {
    this.gson = gson;
    this.adapter = adapter;
  }

  @Override public T convert(ResponseBody value) throws IOExcept
ion {
    JsonReader jsonReader = gson.newJsonReader(value.charStream(
));
    try {
      return adapter.read(jsonReader);
    } finally {
      value.close();
    }
  }
}
```

根据目标类型，利用 `Gson#getAdapter` 获取相应的 adapter，转换时利用 Gson 的 API 即可。

一、EventBus简介

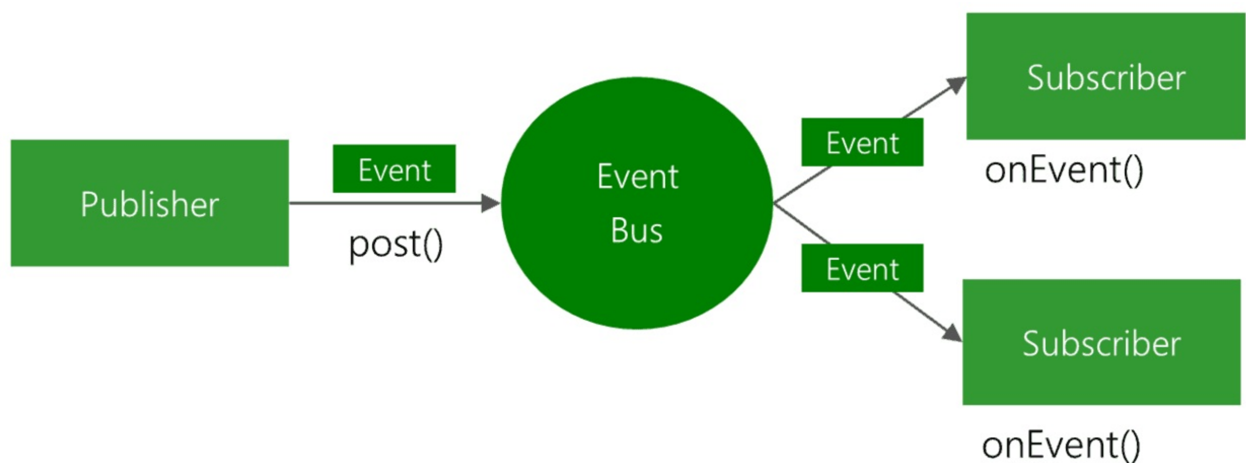
EventBus是一个Android端优化的publish/subscribe消息总线，简化了应用程序内各组件间、组件与后台线程间的通信。

作为一个消息总线主要有三个组成部分：

事件（Event）：可以是任意类型的对象。通过事件的发布者将事件进行传递。

事件订阅者（Subscriber）：接收特定的事件。

事件发布者（Publisher）：用于通知 Subscriber 有事件发生。可以在任意线程任意位置发送事件。



上图解释了整个EventBus的大概工作流程：事件的发布者（Publisher）将事件（Event）通过post()方法发送。EventBus内部进行处理，找到订阅了该事件（Event）的事件订阅者（Subscriber）。然后该事件的订阅者（Subscriber）通过onEvent()方法接收事件进行相关处理（关于onEvent()在EventBus 3.0中有改动，下面详细说明）。

二、EventBus的简单使用

1、把EventBus依赖到项目

build.gradle添加引用

```
compile 'de.greenrobot:eventbus:3.0.0-beta1'
```

2、构造事件（Event）对象。也就是发送消息类 每一个消息类，对应一种事件。这里我们定义两个消息发送类。后面讲解具体作用。

```
public class NewsEvent {
    private String message;

    public NewsEvent(String message) {
        this.message = message;
    }

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }
}
```

```
public class ToastEvent {
    private String content;

    public ToastEvent(String content) {
        this.content = content;
    }

    public String getContent() {
        return content;
    }

    public void setContent(String content) {
        this.content = content;
    }
}
```

3、注册/解除事件订阅（Subscriber）

```
EventBus.getDefault().register(this);//注册事件 其中this代表订阅者
```

具体注册了对什么事件的订阅，这个需要onEvent()方法来说明。在EventBus 3.0之前，onEvent()方法是用来接收指定事件（Event）类型对象，然后进行相关处理操作。在EventBus 3.0之后，onEvent()方法可以自定义方法名，不过要加入注解@Subscribe。

```
@Subscribe
    public void onToastEvent(ToastEvent event){
        Toast.makeText(MainActivity.this,event.getContent(),Toas
t.LENGTH_SHORT).show();
    }
```

通过register(this)来表示该订阅者进行了订阅，通过onToastEvent(ToastEvent event)表示指定对事件ToastEvent的订阅。到这里订阅就完成了。

需要注意的是：一般在onCreate()方法中进行注册订阅。在onDestory()方法中进行解除订阅。

```
@Override
    protected void onDestroy() {
        super.onDestroy();
        EventBus.getDefault().unregister(this);
    }
```

4 、发送消息 订阅已经完成，那么便可以发送订阅了。

```
EventBus.getDefault().post(new ToastEvent("Toast,发个提示，祝大家新
年快乐！"));
```

那么onToastEvent(ToastEvent event)会收到事件，并弹出提示。

EventBus的基础使用流程就是这样的。

其实，EventBus还有好多其他的功能。下面我们一个个介绍。

三、EventBus的进阶使用

1.线程模式ThreadMode

当你接收的的事件后，如果处于非UI线程，你要更新UI怎么办？如果处于UI线程，你要进行耗时操作，怎么办?等等其他情况，通过ThreadMode统统帮你解决。

用法展示：

```
@Subscribe(threadMode = ThreadMode.MainThread)
    public void  onNewsEvent(NewsEvent event){
        String message = event.getMessage();
        mTv_message.setText(message);
    }
```

使用起来很简单，通过 `@Subscribe(threadMode = ThreadMode.MainThread)` 即可指定。 下面具体介绍下ThreadMode。

关于ThreadMode，一共有四种模式PostThread，PostThread，BackgroundThread以及Async。

PostThread：事件的处理在和事件的发送在相同的进程，所以事件处理时间不应太长，不然影响事件的发送线程。

MainThread: 事件的处理会在UI线程中执行。事件处理时间不能太长，这个不用说的，长了会ANR的。

BackgroundThread：如果事件是在UI线程中发布出来的，那么事件处理就会在子线程中运行，如果事件本来就是子线程中发布出来的，那么事件处理直接在该子线程中执行。所有待处理事件会被加到一个队列中，由对应线程依次处理这些事件，如果某个事件处理时间太长，会阻塞后面的事件的派发或处理。

Async：事件处理会在单独的线程中执行，主要用于在后台线程中执行耗时操作，每个事件会开启一个线程。

2.priority事件优先级

事件的优先级类似广播的优先级，优先级越高优先获得消息。 用法展示：

```
@Subscribe(priority = 100)
    public void onToastEvent(ToastEvent event){
        Toast.makeText(MainActivity.this,event.getContent(),Toas
t.LENGTH_SHORT).show();
    }
```

当多个订阅者（Subscriber）对同一种事件类型进行订阅时，即对应的事件处理方法中接收的事件类型一致，则优先级高（**priority** 设置的值越大），则会先接收事件进行处理；优先级低（**priority** 设置的值越小），则会后接收事件进行处理。

除此之外，EventBus也可以终止对事件继续传递的功能。 用法展示：

```
 @Subscribe(priority = 100)
    public void onToastEvent(ToastEvent event){
        Toast.makeText(MainActivity.this,event.getContent(),Toas
t.LENGTH_SHORT).show();
        EventBus.getDefault().cancelEventDelivery(event);
    }
```

这样其他优先级比100低，并且订阅了该事件的订阅者就会接收不到该事件。

3.EventBus黏性事件

EventBus除了普通事件也支持粘性事件。可以理解成：订阅在发布事件之后，但同样可以收到事件。订阅/解除订阅和普通事件一样，但是处理订阅的方法有所不同，需要注解中添加sticky = true。 用法展示：

```
@Subscribe(priority = 100,sticky = true)
    public void onToastEvent(ToastEvent event){
        Toast.makeText(MainActivity.this,event.getContent(),Toas
t.LENGTH_SHORT).show();
        EventBus.getDefault().cancelEventDelivery(event);
    }
```

这样，假设一个ToastEvent 的事件已经发布，此时还没有注册订阅。当设置了sticky = true，在ToastEvent 的事件发布后，进行注册。依然能够接收到之前发布的事件。

不过这个时候，发布事件的方式就改变了。

```
EventBus.getDefault().postSticky(new ToastEvent("Toast,发个提示，
祝大家新年快乐！"));
```

我们如果不再需要该粘性事件我们可以移除

```
EventBus.getDefault().removeStickyEvent(ToastEvent.class);
```

或者调用移除所有粘性事件

```
EventBus.getDefault().removeAllStickyEvents();
```

4.EventBus配置

EventBus在2.3版本中添加了EventBuilder去配置EventBus的各方各面。

比如：如何去构建一个在发布事件时没有订阅者时保持沉默的EventBus。

```
EventBus eventBus = EventBus.builder()
.logNoSubscriberMessages(false)
.sendNoSubscriberEvent(false)
.build();
```

通过上述设置，当一个事件没有订阅者时，不会输出log信息，也不会发布一条默认信息。

配置默认的EventBus实例，使用EventBus.getDefault()是一个简单的方法。获取一个单例的EventBus实例。EventBusBuilder也允许使用installDefaultEventBus方法去配置默认的EventBus实例。

注意：不同的**EventBus** 的对象的数据是不共享的。通过一个**EventBus** 对象去发布事件，只有通过同一个**EventBus** 对象订阅事件，才能接收该事件。所以以上使用**EventBus.getDefault()**获得的都是同一个实例。

四、源码解析

注册订阅

```
EventBus.getDefault().register(this);
```

事件处理

```
@Subscribe(threadMode = ThreadMode.MainThread)
    public void  onNewsEvent(NewsEvent event){
        String message = event.getMessage();
        mTv_message.setText(message);
    }
```

发布事件

```
 EventBus.getDefault().post(new NewsEvent("我是来自SecondActivity
的消息，你好！"));
```

以上是EventBus的基本使用。我们先从getDefault说起。

getDefault()

```
static volatile EventBus defaultInstance;
public static EventBus getDefault() {
        if (defaultInstance == null) {
            synchronized (EventBus.class) {
                if (defaultInstance == null) {
                    defaultInstance = new EventBus();
                }
            }
        }
        return defaultInstance;
    }
```

通过上述代码可以得知，getDefault()中通过双检查锁（DCL）机制实现了EventBus的单例机制，获得了一个默认配置的EventBus对象。 下面我们继续看register()方法。

register()

在了解register()之前，我们先要了解一下EventBus中的几个关键的成员变量。方便对下面内容的理解。

```
/** Map<订阅事件, 订阅该事件的订阅者集合> */
private final Map<Class<?>, CopyOnWriteArrayList<Subscription>>
subscriptionsByEventType;

/** Map<订阅者, 订阅事件集合> */
private final Map<Object, List<Class<?>>> typesBySubscriber;

/** Map<订阅事件类类型,订阅事件实例对象>. */
private final Map<Class<?>, Object> stickyEvents;
```

下面看具体的register()中执行的代码。

```
public void register(Object subscriber) {
        //订阅者类型
        Class<?> subscriberClass = subscriber.getClass();
        //判断该类是不是匿名类，如果是匿名累要使用反射
        boolean forceReflection = subscriberClass.isAnonymousCla
ss();
        //获取订阅者全部的响应函数信息（即上面的onNewsEvent()之类的方法）

        List<SubscriberMethod> subscriberMethods =
                subscriberMethodFinder.findSubscriberMethods(sub
scriberClass, forceReflection);
        //循环每一个事件响应函数，执行 subscribe()方法，更新订阅相关信息
        for (SubscriberMethod subscriberMethod : subscriberMetho
ds) {
            subscribe(subscriber, subscriberMethod);
        }
    }
```

由此可见，register()第一步获取订阅者的类类型. 第二步,通过SubscriberMethodFinder类来解析订阅者类,获取所有的响应函数集合. 第三步,遍历订阅函数,执行 subscribe()方法，更新订阅相关信息。 关于subscriberMethodFinder这里就不介绍了。先跟着线索，继续看subscribe()方法。subscribe 函数分三步。

第一步：

```
        //获取订阅的事件类型
        Class<?> eventType = subscriberMethod.eventType;
       //获取订阅该事件的订阅者集合
        CopyOnWriteArrayList<Subscription> subscriptions = subsc
riptionsByEventType.get(eventType);
      //把通过register()订阅的订阅者包装成Subscription 对象
        Subscription newSubscription = new Subscription(subscrib
er, subscriberMethod);
        //订阅者集合为空，创建新的集合，并把newSubscription 加入
        if (subscriptions == null) {
            subscriptions = new CopyOnWriteArrayList<Subscriptio
n>();
            subscriptionsByEventType.put(eventType, subscription
s);
        } else {
            //集合中已经有该订阅者，抛出异常。不能重复订阅
            if (subscriptions.contains(newSubscription)) {
                throw new EventBusException("Subscriber " + subs
criber.getClass() + " already registered to event "
                        + eventType);
            }
        }
       //把新的订阅者按照优先级加入到订阅者集合中。
        synchronized (subscriptions) {
            int size = subscriptions.size();
            for (int i = 0; i <= size; i++) {
                if (i == size || subscriberMethod.priority > sub
scriptions.get(i).subscriberMethod.priority) {
                    subscriptions.add(i, newSubscription);
                    break;
                }
            }
        }
```

第二步：

```
        //根据订阅者，获得该订阅者订阅的事件类型集合
        List<Class<?>> subscribedEvents = typesBySubscriber.get(
subscriber);
       //如果事件类型集合为空，创建新的集合，并加入新订阅的事件类型。
        if (subscribedEvents == null) {
            subscribedEvents = new ArrayList<Class<?>>();
            typesBySubscriber.put(subscriber, subscribedEvents);
        }
       //如果事件类型集合不为空，加入新订阅的事件类型
        subscribedEvents.add(eventType);
```

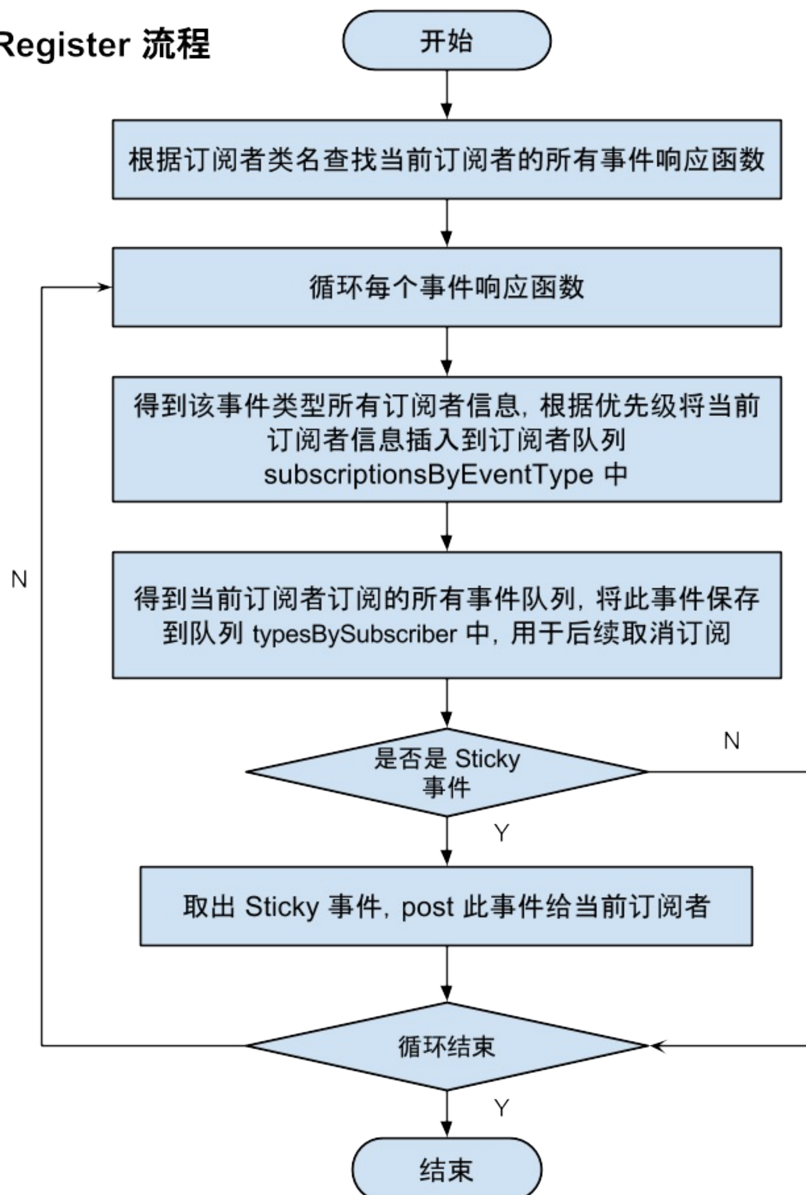
第三步：

```
//该事件是stick=true。
if (subscriberMethod.sticky) {
            //响应订阅事件的父类事件
            if (eventInheritance) {
                Set<Map.Entry<Class<?>, Object>> entries = stick
yEvents.entrySet();
                //循环获得每个stickyEvent事件
                for (Map.Entry<Class<?>, Object> entry : entries
) {
                    Class<?> candidateEventType = entry.getKey()
;
                    //是该类的父类
                    if (eventType.isAssignableFrom(candidateEven
tType)) {
                     //该事件类型最新的事件发送给当前订阅者。
                        Object stickyEvent = entry.getValue();
                        checkPostStickyEventToSubscription(newSu
bscription, stickyEvent);
                    }
                }
            } else {
                Object stickyEvent = stickyEvents.get(eventType)
;
                checkPostStickyEventToSubscription(newSubscripti
on, stickyEvent);
            }
        }
```

由此可见，第一步：通过subscriptionsByEventType得到该事件类型所有订阅者信息队列，根据优先级将当前订阅者信息插入到订阅者队列subscriptionsByEventType中；

第二步：在typesBySubscriber中得到当前订阅者订阅的所有事件队列，将此事件保存到队列typesBySubscriber中，用于后续取消订阅； 第三步：检查这个事件是否是 Sticky 事件，如果是则从stickyEvents事件保存队列中取出该事件类型最后一个事件发送给当前订阅者。

到此，便完成了订阅功能。下面是订阅的具体流程图：


Register 流程
开始
根据订阅者类名查找当前订阅者的所有事件响应函数
循环每个事件响应函数
得到该事件类型所有订阅者信息，根据优先级将当前订阅者信息插入到订阅者队列 subscriptionsByEventType 中
得到当前订阅者订阅的所有事件队列，将此事件保存到队列 typesBySubscriber 中，用于后续取消订阅
是否是 Sticky 事件
N
Y
取出 Sticky 事件，post 此事件给当前订阅者
循环结束
N
Y
结束

unregister()

```
public synchronized void unregister(Object subscriber) {
    // 获取该订阅者所有的订阅事件类类型集合.
    List<Class<?>> subscribedTypes = typesBySubscriber.get(subsc
riber);
    if (subscribedTypes != null) {
        for (Class<?> eventType : subscribedTypes) {
            unsubscribeByEventType(subscriber, eventType);
        }
        // 从typesBySubscriber删除该<订阅者对象,订阅事件类类型集合>
        typesBySubscriber.remove(subscriber);
    } else {
        Log.e("EventBus", "Subscriber to unregister was not regi
stered before: "
                + subscriber.getClass());
    }
}
```

```
private void unsubscribeByEventType(Object subscriber, Class<?>
eventType) {
    // 获取订阅事件对应的订阅者信息集合.
    List<Subscription> subscriptions = subscriptionsByEventType.
get(eventType);
    if (subscriptions != null) {
        int size = subscriptions.size();
        for (int i = 0; i < size; i ++) {
            Subscription subscription = subscriptions.get(i);
            // 从订阅者集合中删除特定的订阅者.
            if (subscription.subscriber == subscriber) {
                subscription.active = false;
                subscriptions.remove(i);
                i --;
                size --;
            }
        }
    }
}
```

unregister()方法比较简单，主要完成了subscriptionsByEventType以及typesBySubscriber两个集合的同步。

post()

```
public void post(Object event) {
    // 获取当前线程的Posting状态.
    PostingThreadState postingState = currentPostingThreadState.
get();
    // 获取当前线程的事件队列.
    List<Object> eventQueue = postingState.eventQueue;
    //将当前事件添加到其事件队列
    eventQueue.add(event);
    //判断新加入的事件是否在分发中
    if (!postingState.isPosting) {
        postingState.isMainThread = Looper.getMainLooper() == Lo
oper.myLooper();
        postingState.isPosting = true;
        if (postingState.canceled) {
            throw new EventBusException("Internal error. Abort s
tate was not reset");
        }
        try {
            // 循环处理当前线程eventQueue中的每一个event对象.
            while (!eventQueue.isEmpty()) {
                postSingleEvent(eventQueue.remove(0), postingSta
te);
            }
        } finally {
            // 处理完知乎重置postingState一些标识信息.
            postingState.isPosting = false;
            postingState.isMainThread = false;
        }
    }
}
```

post 函数会首先得到当前线程的 post 信息PostingThreadState，其中包含事件队列，将当前事件添加到其事件队列中，然后循环调用 postSingleEvent 函数发布队列中的每个事件。

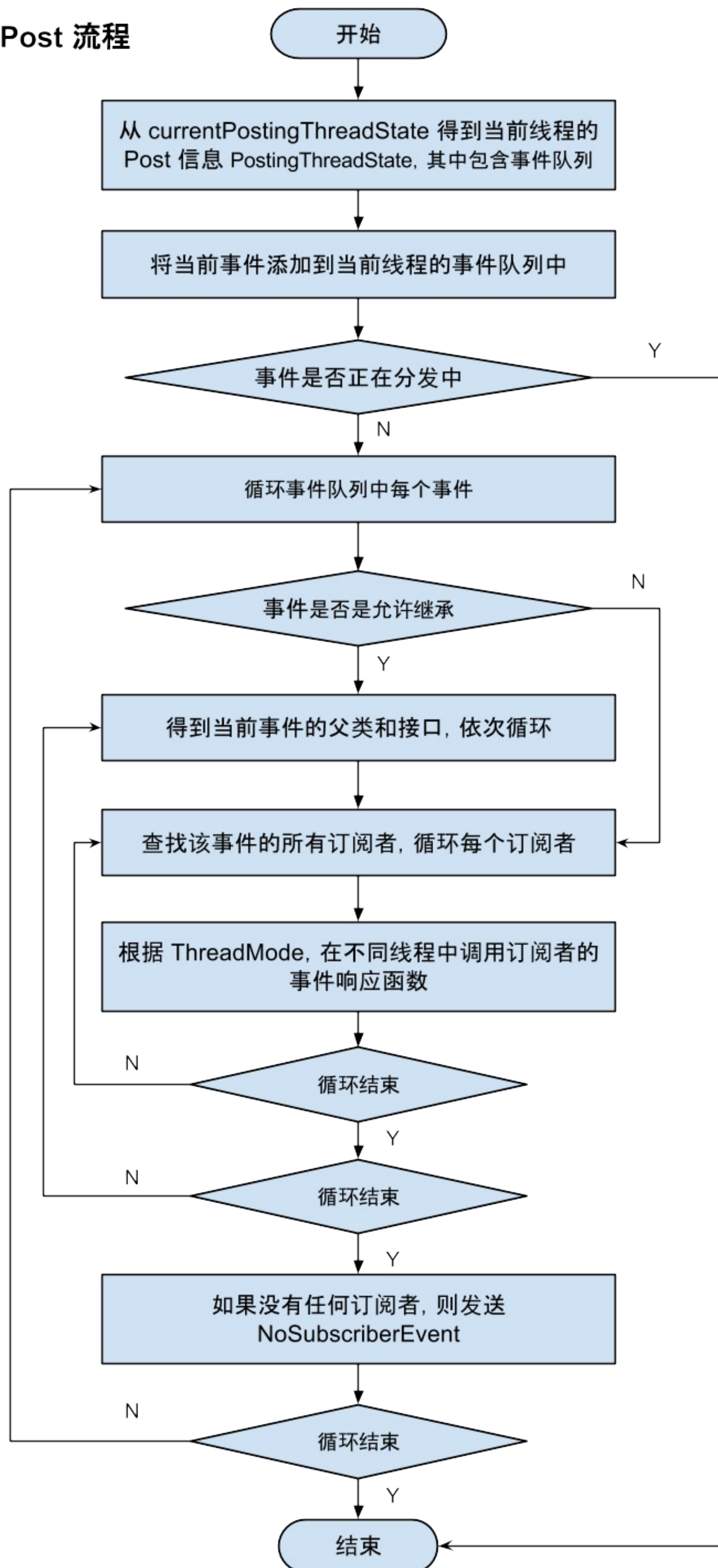
```
private void postSingleEvent(Object event, PostingThreadState po
stingState) {
   //分发事件的类型
    Class<?> eventClass = event.getClass();
    boolean subscriptionFound = false;
   //响应订阅事件的父类事件
    if (eventInheritance) {
        //找出当前订阅事件类类型eventClass的所有父类的类类型和其实现的接
口的类类型
        List<Class<?>> eventTypes = lookupAllEventTypes(eventCla
ss);
        int countTypes = eventTypes.size();
        for (int h = 0; h < countTypes; h ++) {
            Class<?> clazz = eventTypes.get(h);
            //发布每个事件到每个订阅者
            subscriptionFound |= postSingleEventForEventType(eve
nt, postingState, clazz);
        }
    } else {
        subscriptionFound = postSingleEventForEventType(event, p
ostingState, eventClass);
    }
....................................
}
```

调用 postSingleEventForEventType 函数发布每个事件到每个订阅者。

```
private boolean postSingleEventForEventType(Object event, Postin
gThreadState postingState,
                                            Class<?> eventClass)
{
    CopyOnWriteArrayList<Subscription> subscriptions;
    synchronized (this) {
        // 获取订阅事件类类型对应的订阅者信息集合.(register函数时构造的集
合)
        subscriptions = subscriptionsByEventType.get(eventClass)
;
    }

    if (subscriptions != null && !subscriptions.isEmpty()) {
        for (Subscription subscription : subscriptions) {
            postingState.event = event;
            postingState.subscription = subscription;
            boolean aborted = false;
            try {
                // 发布订阅事件给订阅函数
                postToSubscription(subscription, event, postingS
tate.isMainThread);
                aborted = postingState.canceled;
            } finally {
                postingState.event = null;
                postingState.subscription = null;
                postingState.canceled = false;
            }
            if (aborted) {
                break;
            }
        }
        return true;
    }
    return false;
}
```

调用 postToSubscription 函数向每个订阅者发布事件。 postToSubscription 函数中会判断订阅者的 ThreadMode，从而决定在什么 Mode 下执行事件响应函数。这里就不贴源码了。后续还牵着到反射以及线程调度问题，这里就不展开了。 以上就是post的流程，下面是具体的post的流程图。


Post 流程
开始
从 currentPostingThreadState 得到当前线程的 Post 信息 PostingThreadState，其中包含事件队列
将当前事件添加到当前线程的事件队列中
事件是否正在分发中
Y
N
循环事件队列中每个事件
事件是否是允许继承
N
Y
得到当前事件的父类和接口，依次循环
查找该事件的所有订阅者，循环每个订阅者
根据 ThreadMode，在不同线程中调用订阅者的事件响应函数
N
循环结束
Y
N
循环结束
Y
如果没有任何订阅者，则发送 NoSubscriberEvent
N
循环结束
Y
结束

时间： 2018-05-12 01:44:07

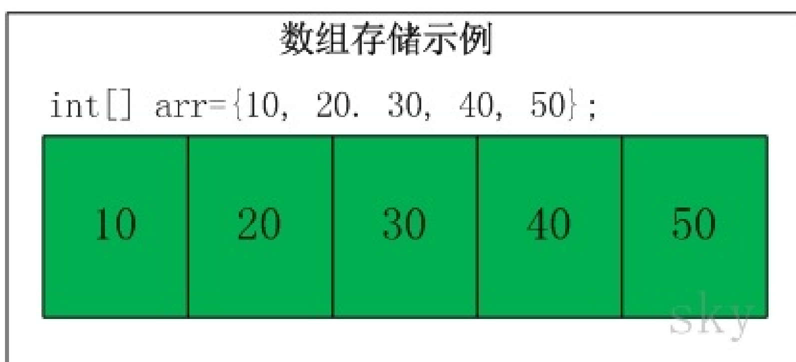
概要

线性表是一种线性结构，它是具有相同类型的n(n≥0)个数据元素组成的有限序列。本章先介绍线性表的几个基本组成部分：数组、单向链表、双向链表。

数组

数组有上界和下界，数组的元素在上下界内是连续的。

存储10,20,30,40,50的数组的示意图如下：

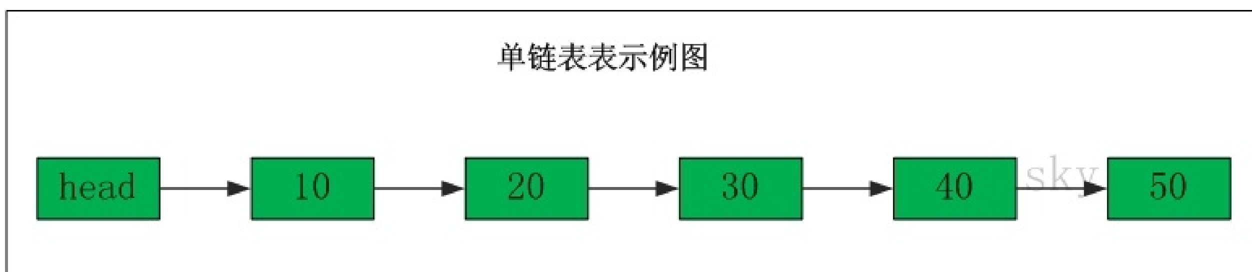


数组的特点是：数据是连续的；随机访问速度快。 数组中稍微复杂一点的是多维数组和动态数组。对于C语言而言，多维数组本质上也是通过一维数组实现的。至于动态数组，是指数组的容量能动态增长的数组；对于C语言而言，若要提供动态数组，需要手动实现；而对于C++而言，STL提供了Vector；对于Java而言，Collection集合中提供了ArrayList和Vector。

单向链表

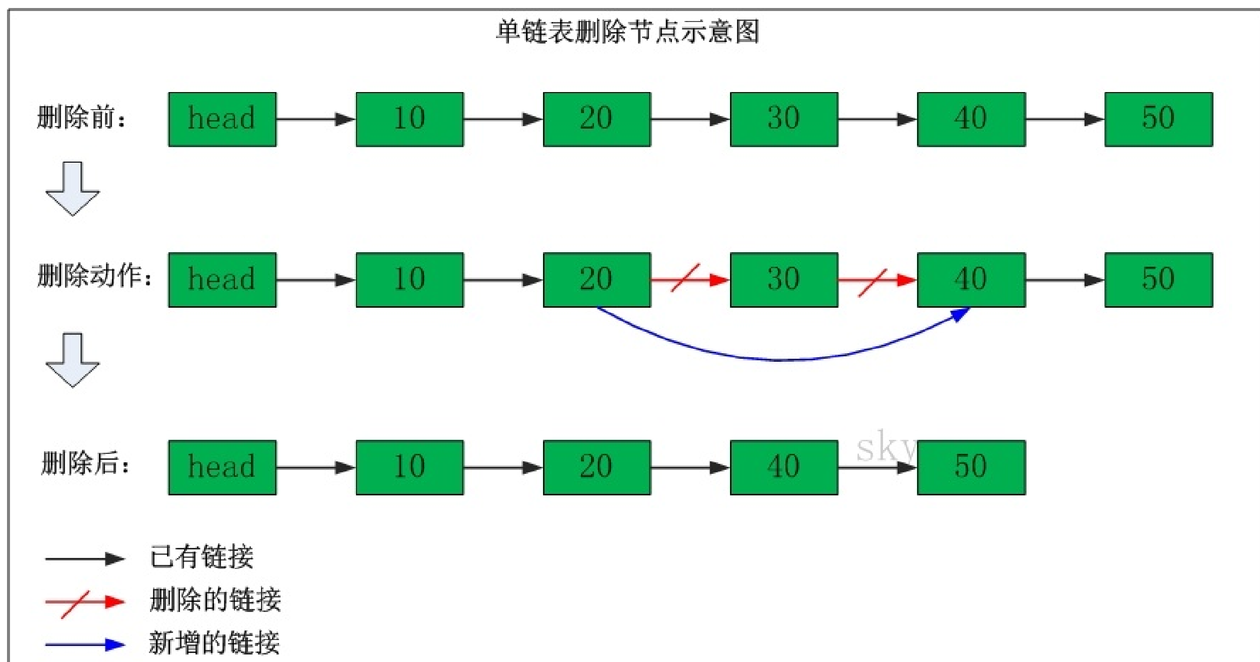
单向链表(单链表)是链表的一种，它由节点组成，每个节点都包含下一个节点的指针。

单链表的示意图如下：



表头为空，表头的后继节点是"节点10"(数据为10的节点)，"节点10"的后继节点是"节点20"(数据为10的节点)，...

单链表删除节点

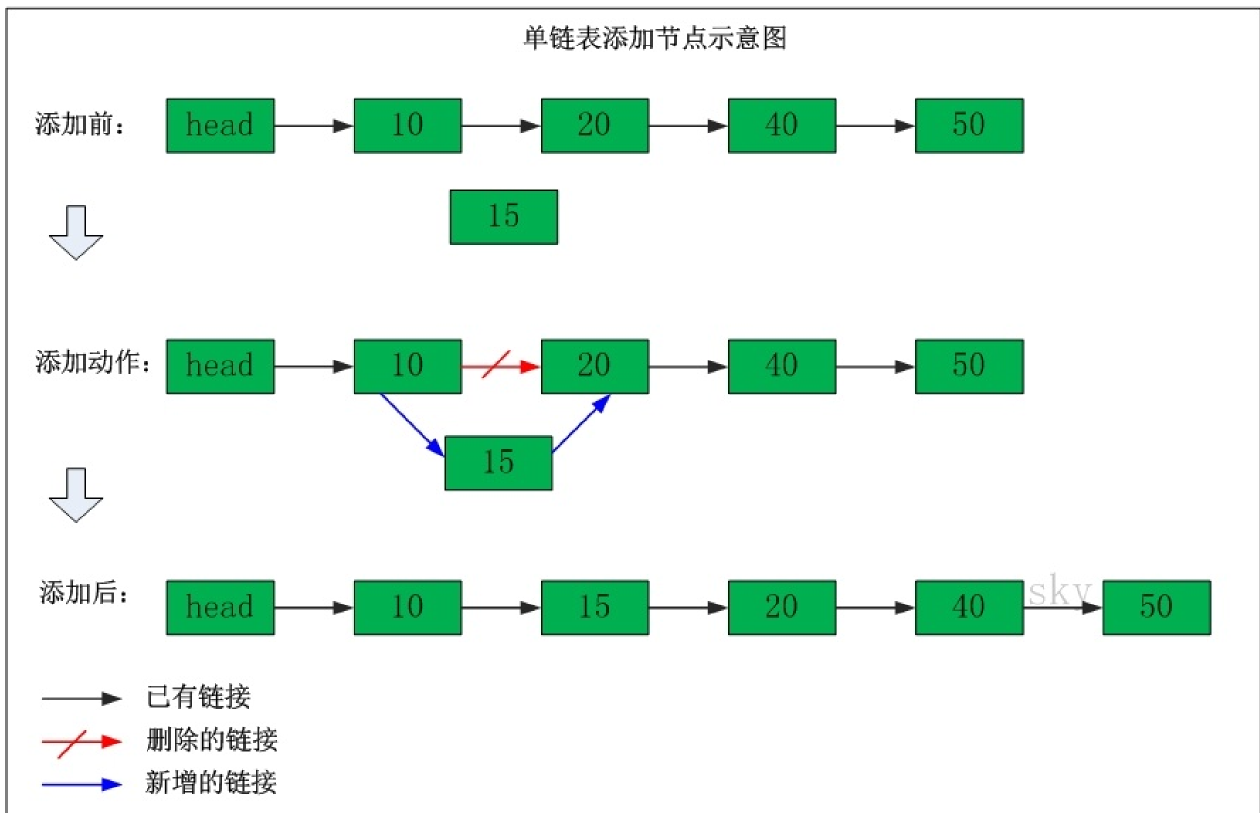


删除"节点30"

删除之前："节点20" 的后继节点为"节点30"，而"节点30" 的后继节点为"节点40"。

删除之后："节点20" 的后继节点为"节点40"。

单链表添加节点



在"节点10"与"节点20"之间添加"节点15"

添加之前："节点10" 的后继节点为"节点20"。

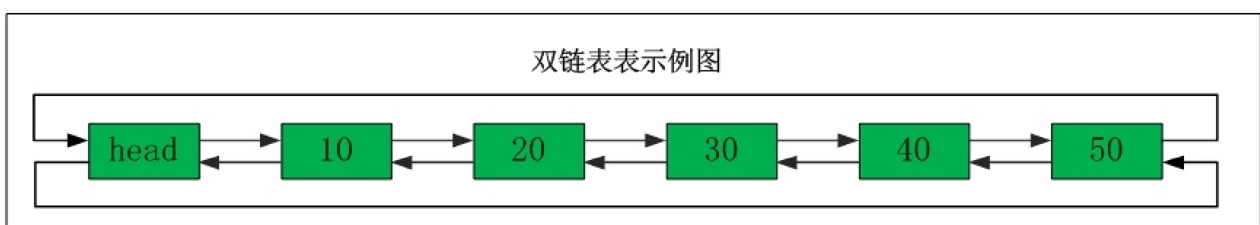
添加之后："节点10" 的后继节点为"节点15"，而"节点15" 的后继节点为"节点20"。

单链表的特点是：节点的链接方向是单向的；相对于数组来说，单链表的的随机访问速度较慢，但是单链表删除/添加数据的效率很高。

双向链表

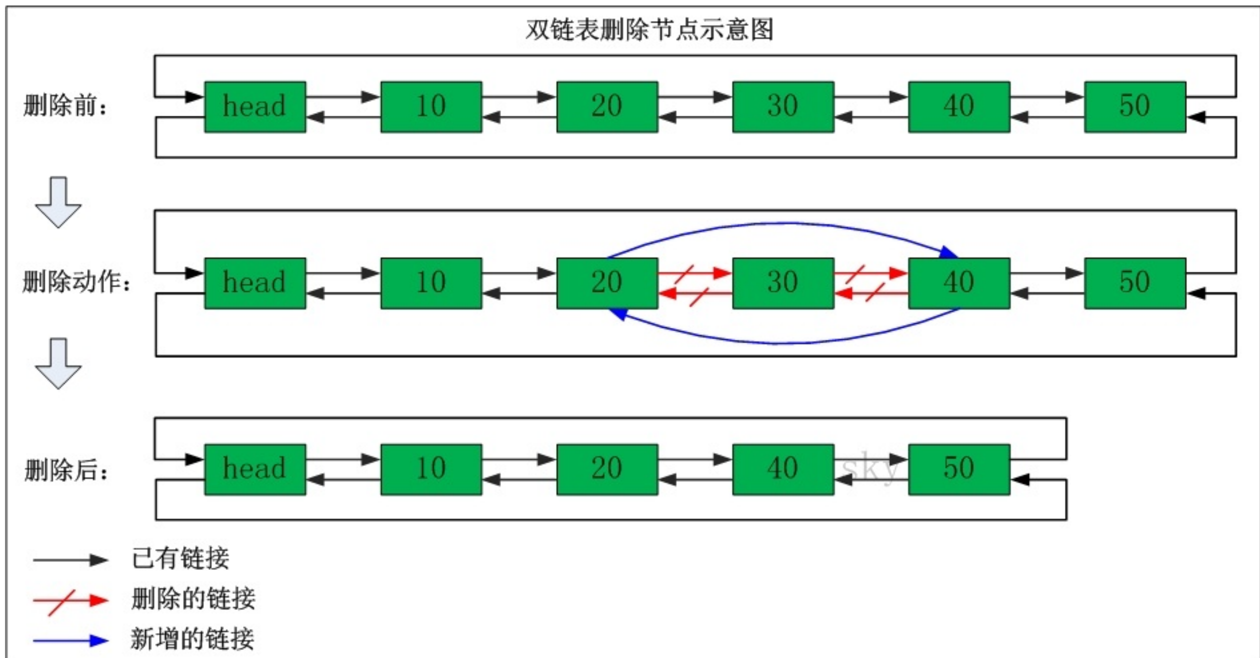
双向链表(双链表)是链表的一种。和单链表一样，双链表也是由节点组成，它的每个数据结点中都有两个指针，分别指向直接后继和直接前驱。所以，从双向链表中的任意一个结点开始，都可以很方便地访问它的前驱结点和后继结点。一般我们都构造双向循环链表。

双链表的示意图如下：



表头为空，表头的后继节点为"节点10"(数据为10的节点)；"节点10"的后继节点是"节点20"(数据为10的节点)，"节点20"的前继节点是"节点10"；"节点20"的后继节点是"节点30"，"节点30"的前继节点是"节点20"；...；末尾节点的后继节点是表头。

双链表删除节点

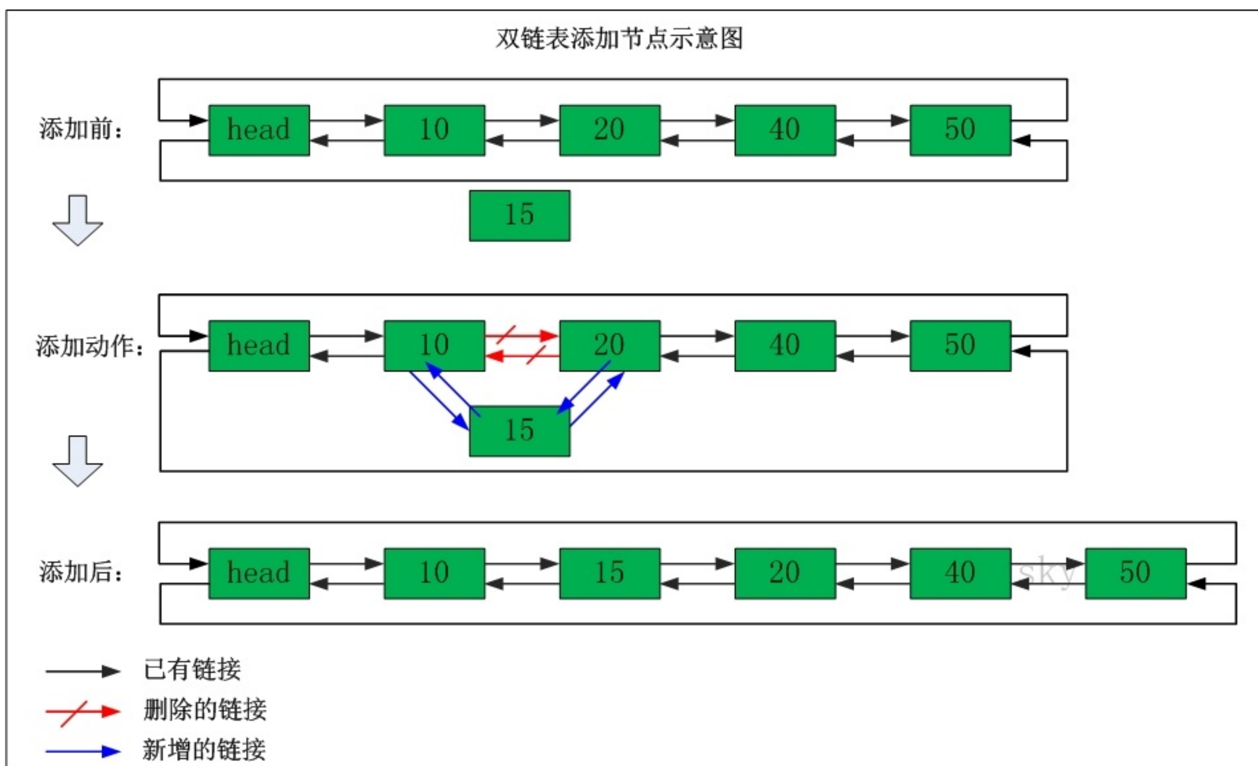


删除"节点30"

删除之前："节点20"的后继节点为"节点30"，"节点30" 的前继节点为"节点20"。"节点30"的后继节点为"节点40"，"节点40" 的前继节点为"节点30"。

删除之后："节点20"的后继节点为"节点40"，"节点40" 的前继节点为"节点20"。

双链表添加节点



在"节点10"与"节点20"之间添加"节点15"

添加之前："节点10"的后继节点为"节点20"，"节点20" 的前继节点为"节点10"。

添加之后："节点10"的后继节点为"节点15"，"节点15" 的前继节点为"节点10"。"节点15"的后继节点为"节点20"，"节点20" 的前继节点为"节点15"。

双链表的Java实现

```
/**
 * Java 实现的双向链表。
 * 注：java自带的集合包中有实现双向链表，路径是:java.util.LinkedList
 *
 * @author skywang
 * @date 2013/11/07
 */
public class DoubleLink<T> {

    // 表头
    private DNode<T> mHead;
    // 节点个数
    private int mCount;
```

```
    // 双向链表“节点”对应的结构体
    private class DNode<T> {
        public DNode prev;
        public DNode next;
        public T value;

        public DNode(T value, DNode prev, DNode next) {
            this.value = value;
            this.prev = prev;
            this.next = next;
        }
    }

    // 构造函数
    public DoubleLink() {
        // 创建“表头”。注意：表头没有存储数据！
        mHead = new DNode<T>(null, null, null);
        mHead.prev = mHead.next = mHead;
        // 初始化“节点个数”为0
        mCount = 0;
    }

    // 返回节点数目
    public int size() {
        return mCount;
    }

    // 返回链表是否为空
    public boolean isEmpty() {
        return mCount==0;
    }

    // 获取第index位置的节点
    private DNode<T> getNode(int index) {
        if (index<0 || index>=mCount)
            throw new IndexOutOfBoundsException();

        // 正向查找
        if (index <= mCount/2) {
            DNode<T> node = mHead.next;
```

```
            for (int i=0; i<index; i++)
                node = node.next;

            return node;
        }

        // 反向查找
        DNode<T> rnode = mHead.prev;
        int rindex = mCount - index -1;
        for (int j=0; j<rindex; j++)
            rnode = rnode.prev;

        return rnode;
    }

    // 获取第index位置的节点的值
    public T get(int index) {
        return getNode(index).value;
    }

    // 获取第1个节点的值
    public T getFirst() {
        return getNode(0).value;
    }

    // 获取最后一个节点的值
    public T getLast() {
        return getNode(mCount-1).value;
    }

    // 将节点插入到第index位置之前
    public void insert(int index, T t) {
        if (index==0) {
            DNode<T> node = new DNode<T>(t, mHead, mHead.next);
            mHead.next.prev = node;
            mHead.next = node;
            mCount++;
            return ;
        }
```

```
        DNode<T> inode = getNode(index);
        DNode<T> tnode = new DNode<T>(t, inode.prev, inode);
        inode.prev.next = tnode;
        inode.prev = tnode;
        mCount++;
        return ;
    }

    // 将节点插入第一个节点处。
    public void insertFirst(T t) {
        insert(0, t);
    }

    // 将节点追加到链表的末尾
    public void appendLast(T t) {
        DNode<T> node = new DNode<T>(t, mHead.prev, mHead);
        mHead.prev.next = node;
        mHead.prev = node;
        mCount++;
    }

    // 删除index位置的节点
    public void del(int index) {
        DNode<T> inode = getNode(index);
        inode.prev.next = inode.next;
        inode.next.prev = inode.prev;
        inode = null;
        mCount--;
    }

    // 删除第一个节点
    public void deleteFirst() {
        del(0);
    }

    // 删除最后一个节点
    public void deleteLast() {
        del(mCount-1);
    }
}
```

一、栈

栈（stack），是一种线性存储结构，它有以下几个特点：

(01) 栈中数据是按照"后进先出（LIFO, Last In First Out）"方式进出栈的。

(02) 向栈中添加/删除数据时，只能从栈顶进行操作。

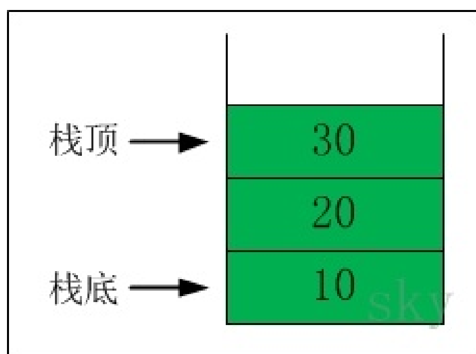
栈通常包括的三种操作：push、peek、pop。

push -- 向栈中添加元素。

peek -- 返回栈顶元素。

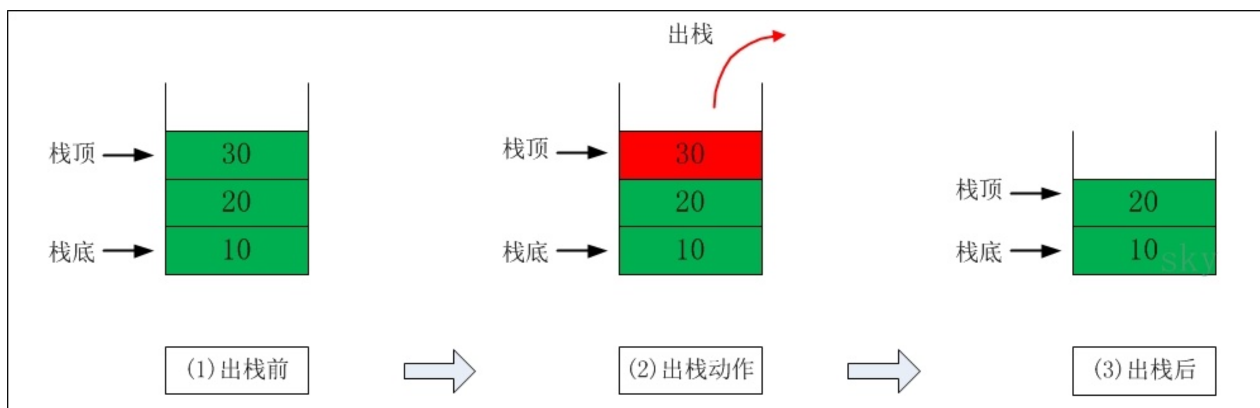
pop -- 返回并删除栈顶元素的操作。

1.栈的示意图



栈中的数据依次是 30 --> 20 --> 10

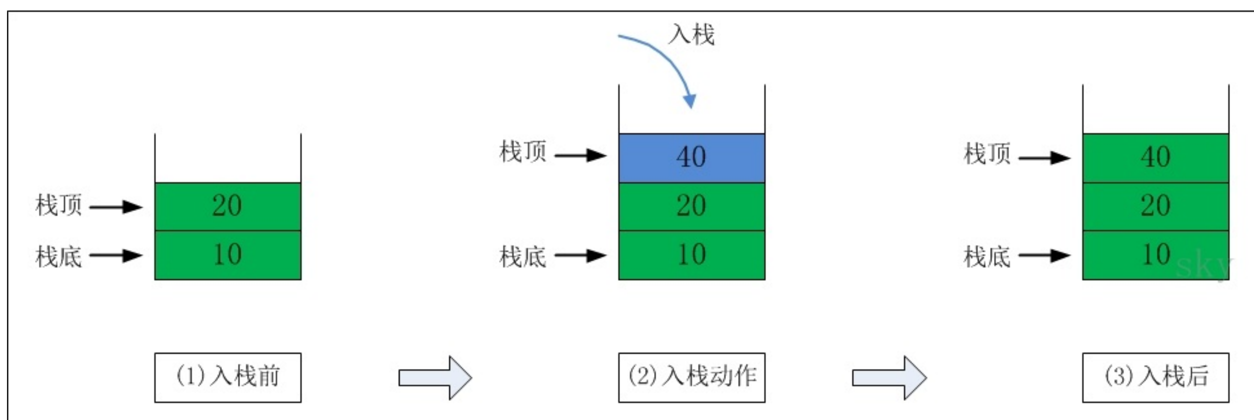
2.出栈



出栈前：栈顶元素是30。此时，栈中的元素依次是 30 --> 20 --> 10

出栈后：30出栈之后，栈顶元素变成20。此时，栈中的元素依次是 20 --> 10

3. 入栈



入栈前：栈顶元素是20。此时，栈中的元素依次是 20 --> 10

入栈后：40入栈之后，栈顶元素变成40。此时，栈中的元素依次是 40 --> 20 --> 10

4.栈的Java实现

JDK包中也提供了"栈"的实现，它就是集合框架中的Stack类。 本部分使用数组实现栈，能存储任意类型的数据。

```
/**
 * Java : 数组实现的栈，能存储任意类型的数据
 *
 * @author skywang
 * @date 2013/11/07
 */
import java.lang.reflect.Array;

public class GeneralArrayStack<T> {

    private static final int DEFAULT_SIZE = 12;
    private T[] mArray;
    private int count;
```

```
    public GeneralArrayStack(Class<T> type) {
        this(type, DEFAULT_SIZE);
    }

    public GeneralArrayStack(Class<T> type, int size) {
        // 不能直接使用mArray = new T[DEFAULT_SIZE];
        mArray = (T[]) Array.newInstance(type, size);
        count = 0;
    }

    // 将val添加到栈中
    public void push(T val) {
        mArray[count++] = val;
    }

    // 返回“栈顶元素值”
    public T peek() {
        return mArray[count-1];
    }

    // 返回“栈顶元素值”，并删除“栈顶元素”
    public T pop() {
        T ret = mArray[count-1];
        count--;
        return ret;
    }

    // 返回“栈”的大小
    public int size() {
        return count;
    }

    // 返回“栈”是否为空
    public boolean isEmpty() {
        return size()==0;
    }

    // 打印“栈”
    public void PrintArrayStack() {
        if (isEmpty()) {
```

```
            System.out.printf("stack is Empty\n");
        }

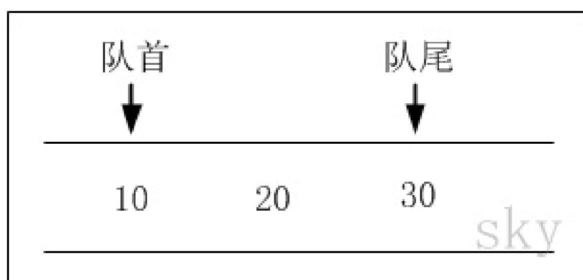
        System.out.printf("stack size()=%d\n", size());

        int i=size()-1;
        while (i>=0) {
            System.out.println(mArray[i]);
            i--;
        }
    }
}
```

二、队列

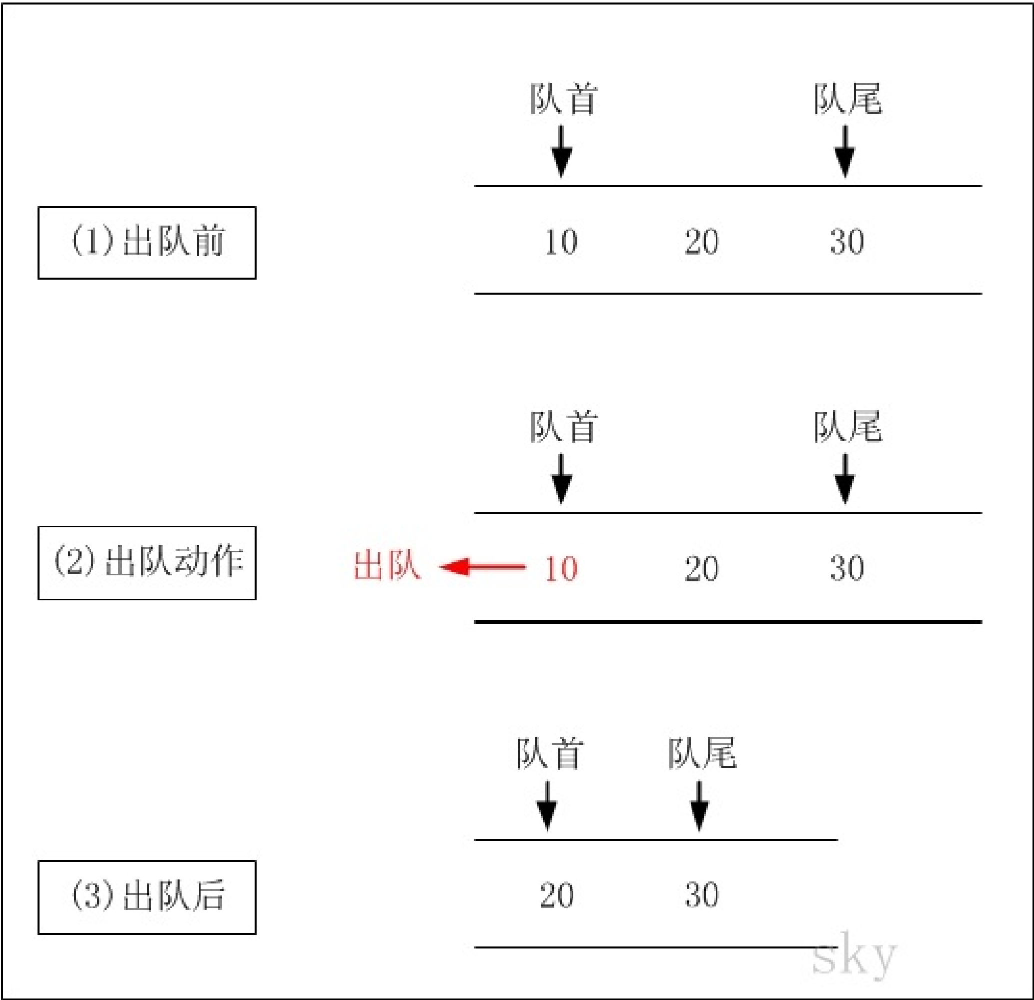
队列（Queue），是一种线性存储结构。它有以下几个特点：(1) 队列中数据是按照"先进先出（FIFO, First-In-First-Out）"方式进出队列的。(2) 队列只允许在"队首"进行删除操作，而在"队尾"进行插入操作。 队列通常包括的两种操作：入队列 和 出队列。

1.队列的示意图



队列中有10，20，30共3个数据。

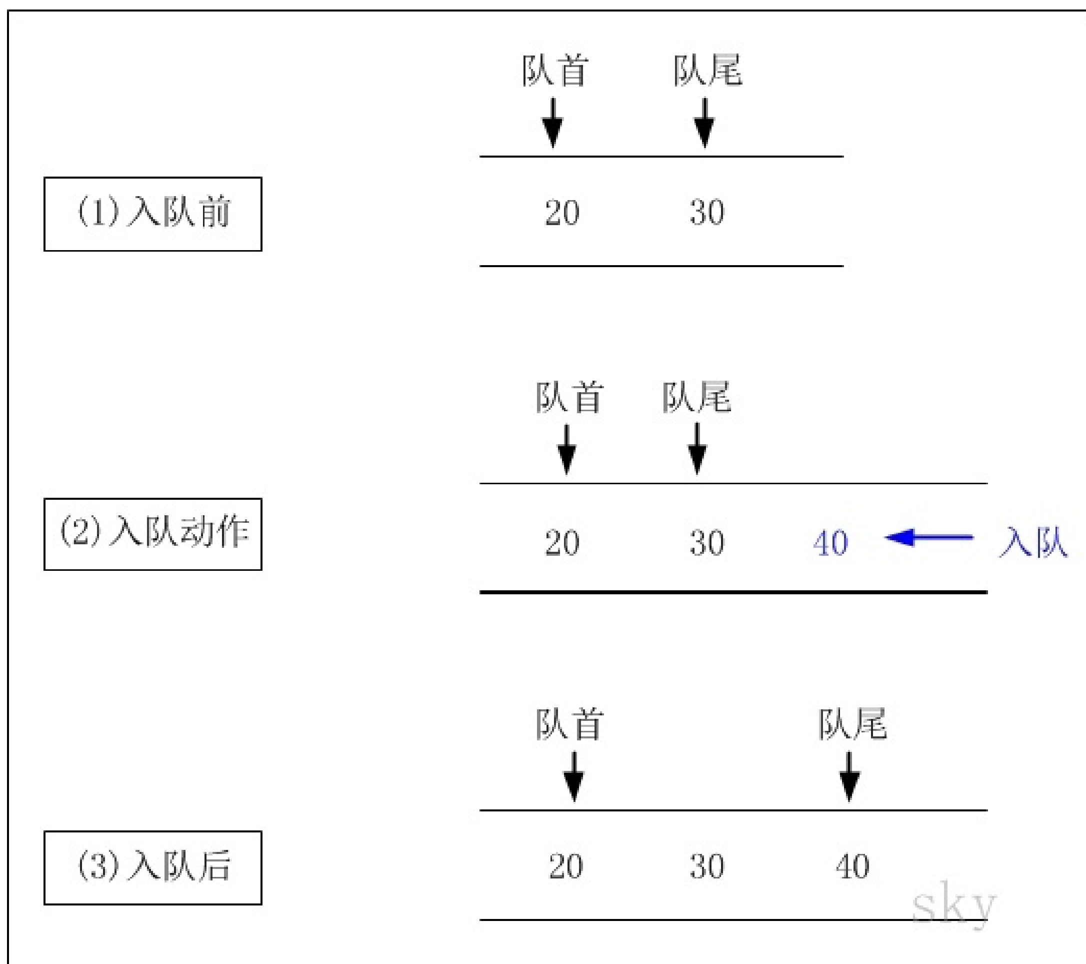
2.出队列



出队列前：队首是10，队尾是30。

出队列后：出队列(队首)之后。队首是20，队尾是30。

3.入队列



入队列前：队首是20，队尾是30。

入队列后：40入队列(队尾)之后。队首是20，队尾是40。

4.队列的Java实现

JDK中的Queue接口就是"队列"，它的实现类也都是队列，用的最多的是LinkedList。本部分使用数组实现队列，能存储任意类型的数据。

```
/**
 * Java : 数组实现“队列”，只能存储int数据。
 *
 * @author skywang
 * @date 2013/11/07
 */
public class ArrayQueue {

    private int[] mArray;
    private int mCount;
```

```
    public ArrayQueue(int sz) {
        mArray = new int[sz];
        mCount = 0;
    }

    // 将val添加到队列的末尾
    public void add(int val) {
        mArray[mCount++] = val;
    }

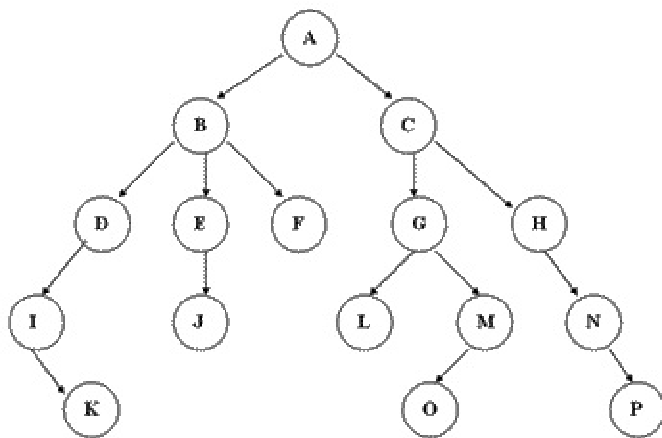
    // 返回“队列开头元素”
    public int front() {
        return mArray[0];
    }

    // 返回“队首元素值”，并删除“队首元素”
    public int pop() {
        int ret = mArray[0];
        mCount--;
        for (int i=1; i<=mCount; i++)
            mArray[i-1] = mArray[i];
        return ret;
    }

    // 返回“栈”的大小
    public int size() {
        return mCount;
    }

    // 返回“栈”是否为空
    public boolean isEmpty() {
        return size()==0;
    }
}
```

一、前言



树作为一种重要的数据结构，即是面试重点考察知识点，也是实际生产应用中可能会灵活应用的一种数据结构。那么其重要性不言而喻，无论是在剑指offer，还是在leetcode中，都有很多关于树的题目。

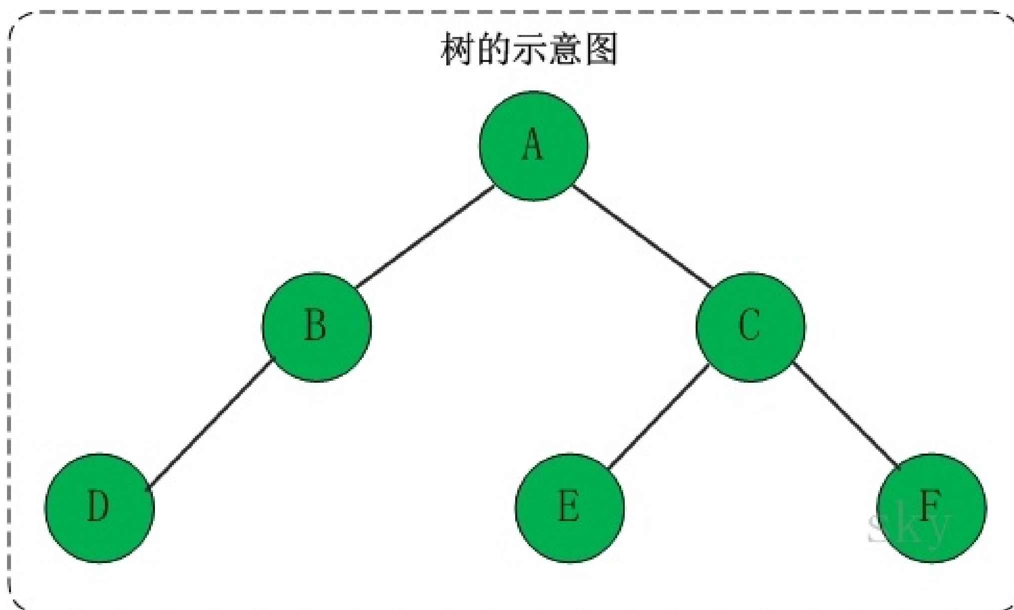
本部分内容主要介绍树的一些基础概念及Java实现，同时介绍一些常见的树，如二叉搜索树，平衡树，红黑树等。

二、目录

一、树的介绍

1.树的定义

树是一种数据结构，它是由n（n>=1）个有限节点组成一个具有层次关系的集合。



把它叫做“树”是因为它看起来像一棵倒挂的树，也就是说它是根朝上，而叶朝下的。它具有以下的特点：

(01) 每个节点有零个或多个子节点；

(02) 没有父节点的节点称为根节点；

(03) 每一个非根节点有且只有一个父节点；

(04) 除了根节点外，每个子节点可以分为多个不相交的子树。

2.树的基本术语

若一个结点有子树，那么该结点称为子树根的"双亲"，子树的根是该结点的"孩子"。有相同双亲的结点互为"兄弟"。一个结点的所有子树上的任何结点都是该结点的后裔。从根结点到某个结点的路径上的所有结点都是该结点的祖先。

结点的度：结点拥有的子树的数目。

叶子：度为零的结点。

分支结点：度不为零的结点。

树的度：树中结点的最大的度。

层次：根结点的层次为1，其余结点的层次等于该结点的双亲结点的层次加1。

树的高度：树中结点的最大层次。

无序树：如果树中结点的各子树之间的次序是不重要的，可以交换位置。

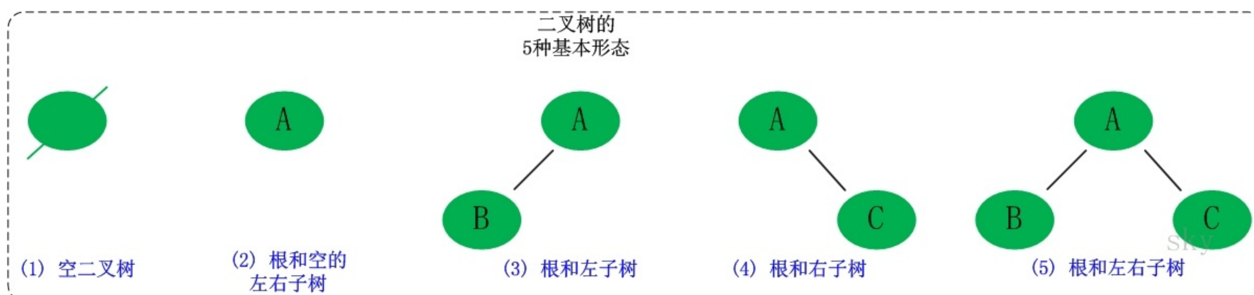
有序树：如果树中结点的各子树之间的次序是重要的, 不可以交换位置。

森林：0个或多个不相交的树组成。对森林加上一个根，森林即成为树；删去根，树即成为森林。

二、二叉树的介绍

1.二叉树的定义

二叉树是每个节点最多有两个子树的树结构。它有五种基本形态：二叉树可以是空集；根可以有空的左子树或右子树；或者左、右子树皆为空。



2. 二叉树的性质

二叉树有以下几个性质：

性质1：二叉树第i层上的结点数目最多为2^{i-1} (i≥1)。

性质2：深度为k的二叉树至多有$2^k - 1$个结点(k≥1)。

性质3：包含n个结点的二叉树的高度至少为$log_2(n+1)$。

性质4：在任意一棵二叉树中，若叶子结点的个数为**n0**，度为2的结点数为**n2**，则**n0=n2+1**。

2.1 性质1：二叉树第i层上的结点数目最多为2^{i-1}(i≥1)

证明：下面用"数学归纳法"进行证明。

(01) 当i=1时，第i层的节点数目为1。因为第1层上只有一个根结点，所以命题成立。

(02) 假设当i>1，第i层的节点数目为2^{i-1}。这个是根据(01)推断出来的！

下面根据这个假设，推断出"第(i+1)层的节点数目为2^{i}"即可。

由于二叉树的每个结点至多有两个孩子，故"第(i+1)层上的结点数目" 最多是 "第i层的结点数目的2倍"。即，第(i+1)层上的结点数目最大值=2×2^{i-1}=2^{i}。

故假设成立，原命题得证！

2.2 性质2：深度为k的二叉树至多有$2^{k}-1$个结点(k≥1)

证明：在具有相同深度的二叉树中，当每一层都含有最大结点数时，其树中结点数最多。利用"性质1"可知，深度为k的二叉树的结点数至多为：

$2^0+2^1+…+2^{k-1}=2^k-1$

故原命题得证！

2.3 性质3：包含n个结点的二叉树的高度至少为$log_2(n+1)$

证明：根据"性质2"可知，高度为h的二叉树最多有$2^{h}-1$个结点。反之，对于包含n个节点的二叉树的高度至少为$log_2(n+1)$。

2.4 性质4：在任意一棵二叉树中，若终端结点的个数为n0，度为2的结点数为n2，则n0=n2+1

证明：因为二叉树中所有结点的度数均不大于2，所以结点总数(记为n)="0度结点数(n0)" + "1度结点数(n1)" + "2度结点数(n2)"。由此，得到等式一。

(等式一) n=n0+n1+n2

另一方面，0度结点没有孩子，1度结点有一个孩子，2度结点有两个孩子，故二叉树中孩子结点总数是：n1+2n2。此外，只有根不是任何结点的孩子。故二叉树中的结点总数又可表示为等式二。

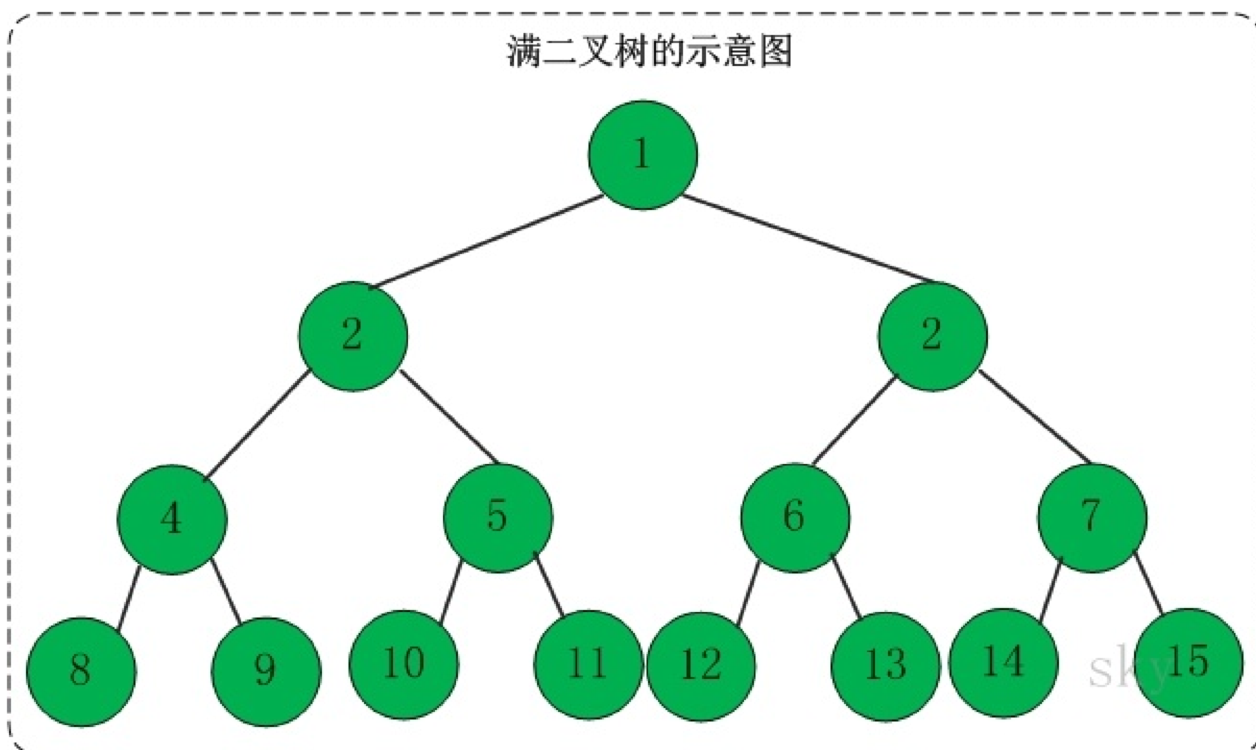
(等式二) n=n1+2n2+1

由(等式一)和(等式二)计算得到：n0=n2+1。原命题得证！

3. 满二叉树，完全二叉树和二叉查找树

3.1 满二叉树

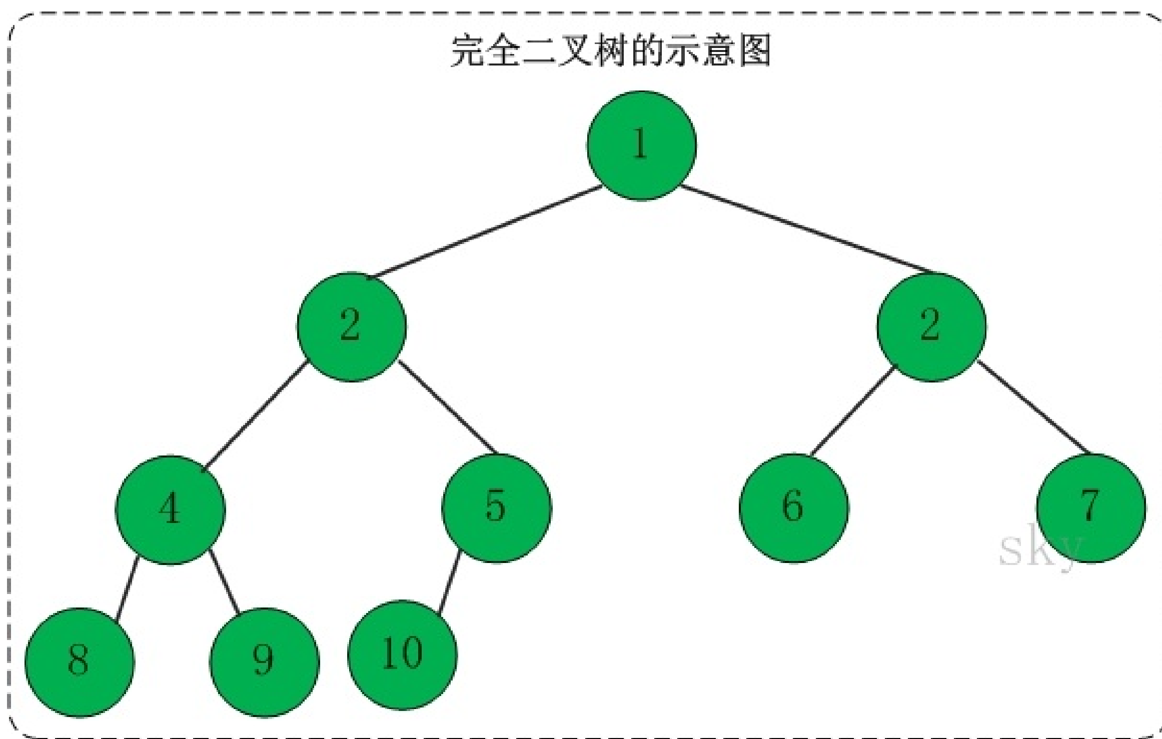
定义：高度为h，并且由$2^{h}-1$个结点的二叉树，被称为满二叉树。



3.2 完全二叉树

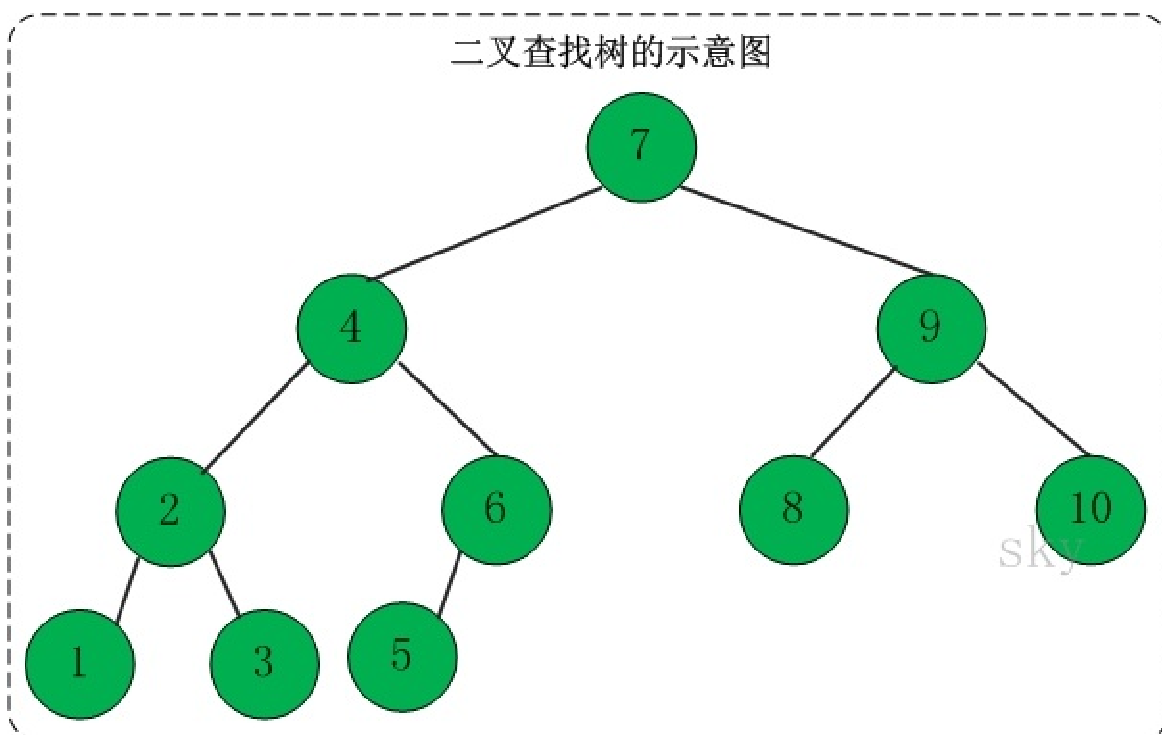
定义：一棵二叉树中，只有最下面两层结点的度可以小于2，并且最下一层的叶结点集中在靠左的若干位置上。这样的二叉树称为完全二叉树。

特点：叶子结点只能出现在最下层和次下层，且最下层的叶子结点集中在树的左部。显然，一棵满二叉树必定是一棵完全二叉树，而完全二叉树未必是满二叉树。



3.3 二叉查找树

定义：二叉查找树(Binary Search Tree)，又被称为二叉搜索树。设x为二叉查找树中的一个结点，x节点包含关键字key，节点x的key值记为key[x]。如果y是x的左子树中的一个结点，则key[y] <= key[x]；如果y是x的右子树的一个结点，则key[y] >= key[x]。



在二叉查找树中：

(01) 若任意节点的左子树不空，则左子树上所有结点的值均小于它的根结点的值；

(02) 任意节点的右子树不空，则右子树上所有结点的值均大于它的根结点的值；

(03) 任意节点的左、右子树也分别为二叉查找树。

(04) 没有键值相等的节点（no duplicate nodes）。

三、二叉查找树的Java实现

1.二叉查找树节点的定义

```
public class BSTree<T extends Comparable<T>> {

    private BSTNode<T> mRoot;    // 根结点

    public class BSTNode<T extends Comparable<T>> {
        T key;                // 关键字(键值)
        BSTNode<T> left;      // 左孩子
        BSTNode<T> right;     // 右孩子
        BSTNode<T> parent;    // 父结点

        public BSTNode(T key, BSTNode<T> parent, BSTNode<T> left
, BSTNode<T> right) {
            this.key = key;
            this.parent = parent;
            this.left = left;
            this.right = right;
        }
    }

        ......
}
```

BSTree是二叉树，它保含了二叉树的根节点mRoot；mRoot是BSTNode类型，而BSTNode是二叉查找树的节点，它是BSTree的内部类。BSTNode包含二叉查找树的几个基本信息：

(01) key -- 它是关键字，是用来对二叉查找树的节点进行排序的。

(02) left -- 它指向当前节点的左孩子。

(03) right -- 它指向当前节点的右孩子。

(04) parent -- 它指向当前节点的父结点。

2.遍历

这里讲解前序遍历、中序遍历、后序遍历3种方式。

2.1 前序遍历

若二叉树非空，则执行以下操作：

(01) 访问根结点；

(02) 先序遍历左子树；

(03) 先序遍历右子树。

前序遍历代码

```
private void preOrder(BSTNode<T> tree) {
    if(tree != null) {
        System.out.print(tree.key+" ");
        preOrder(tree.left);
        preOrder(tree.right);
    }
}

public void preOrder() {
    preOrder(mRoot);
}
```

2.2 中序遍历

若二叉树非空，则执行以下操作：

(01) 中序遍历左子树；

(02) 访问根结点；

(03) 中序遍历右子树。

中序遍历代码

```
private void inOrder(BSTNode<T> tree) {
    if(tree != null) {
        inOrder(tree.left);
        System.out.print(tree.key+" ");
        inOrder(tree.right);
    }
}

public void inOrder() {
    inOrder(mRoot);
}
```

2.3 后序遍历

若二叉树非空，则执行以下操作：

(01) 后序遍历左子树；

(02) 后序遍历右子树；

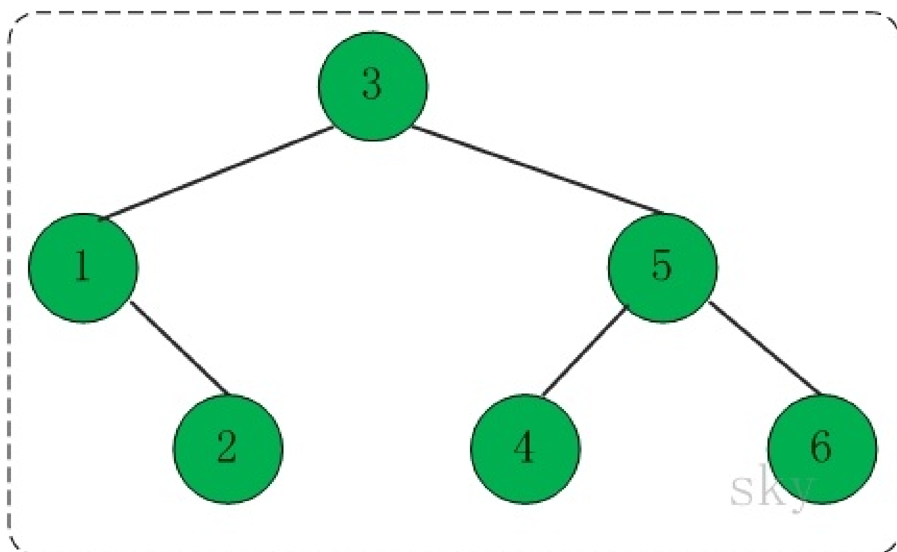
(03) 访问根结点。

后序遍历代码

```
private void postOrder(BSTNode<T> tree) {
    if(tree != null)
    {
        postOrder(tree.left);
        postOrder(tree.right);
        System.out.print(tree.key+" ");
    }
}

public void postOrder() {
    postOrder(mRoot);
}
```

看看下面这颗树的各种遍历方式：



对于上面的二叉树而言，

(01) 前序遍历结果： 3 1 2 5 4 6

(02) 中序遍历结果： 1 2 3 4 5 6

(03) 后序遍历结果： 2 1 4 6 5 3

3. 查找

递归版本的代码

```
/*
 * (递归实现)查找"二叉树x"中键值为key的节点
 */
private BSTNode<T> search(BSTNode<T> x, T key) {
    if (x==null)
        return x;

    int cmp = key.compareTo(x.key);
    if (cmp < 0)
        return search(x.left, key);
    else if (cmp > 0)
        return search(x.right, key);
    else
        return x;
}

public BSTNode<T> search(T key) {
    return search(mRoot, key);
}
```

非递归版本的代码

```
/*
 * (非递归实现)查找"二叉树x"中键值为key的节点
 */
private BSTNode<T> iterativeSearch(BSTNode<T> x, T key) {
    while (x!=null) {
        int cmp = key.compareTo(x.key);

        if (cmp < 0)
            x = x.left;
        else if (cmp > 0)
            x = x.right;
        else
            return x;
    }

    return x;
}

public BSTNode<T> iterativeSearch(T key) {
    return iterativeSearch(mRoot, key);
}
```

4. 最大值和最小值

查找最大值的代码

```
/*
 * 查找最大结点：返回tree为根结点的二叉树的最大结点。
 */
private BSTNode<T> maximum(BSTNode<T> tree) {
    if (tree == null)
        return null;

    while(tree.right != null)
        tree = tree.right;
    return tree;
}

public T maximum() {
    BSTNode<T> p = maximum(mRoot);
    if (p != null)
        return p.key;

    return null;
}
```

查找最小值的代码

```
/*
 * 查找最小结点：返回tree为根结点的二叉树的最小结点。
 */
private BSTNode<T> minimum(BSTNode<T> tree) {
    if (tree == null)
        return null;

    while(tree.left != null)
        tree = tree.left;
    return tree;
}

public T minimum() {
    BSTNode<T> p = minimum(mRoot);
    if (p != null)
        return p.key;

    return null;
}
```

5. 前驱和后继

节点的前驱：是该节点的左子树中的最大节点。

节点的后继：是该节点的右子树中的最小节点。

查找前驱节点的代码

```
/*
 * 找结点(x)的前驱结点。即，查找"二叉树中数据值小于该结点"的"最大结点"。
 */
public BSTNode<T> predecessor(BSTNode<T> x) {
    // 如果x存在左孩子，则"x的前驱结点"为 "以其左孩子为根的子树的最大结点"。

    if (x.left != null)
        return maximum(x.left);

    // 如果x没有左孩子。则x有以下两种可能：
    // (01) x是"一个右孩子"，则"x的前驱结点"为 "它的父结点"。
    // (02) x是"一个左孩子"，则查找"x的最低的父结点，并且该父结点要具有右孩子"，找到的这个"最低的父结点"就是"x的前驱结点"。
    BSTNode<T> y = x.parent;
    while ((y!=null) && (x==y.left)) {//满足条件，不断往上追溯，直到找到右祖先结点
        x = y;
        y = y.parent;
    }

    return y;
}
```

查找后继节点的代码

```
/*
 * 找结点(x)的后继结点。即，查找"二叉树中数据值大于该结点"的"最小结点"。
 */
public BSTNode<T> successor(BSTNode<T> x) {
    // 如果x存在右孩子，则"x的后继结点"为 "以其右孩子为根的子树的最小结点"。

    if (x.right != null)
        return minimum(x.right);

    // 如果x没有右孩子。则x有以下两种可能：
    // (01) x是"一个左孩子"，则"x的后继结点"为 "它的父结点"。
    // (02) x是"一个右孩子"，则查找"x的最低的父结点，并且该父结点要具有左
孩子"，找到的这个"最低的父结点"就是"x的后继结点"。
    BSTNode<T> y = x.parent;
    while ((y!=null) && (x==y.right)) {//满足条件，不断往上追溯，直到
找到右祖先结点
        x = y;
        y = y.parent;
    }

    return y;
}
```

6. 插入

插入节点的代码

```
/*
 * 将结点插入到二叉树中
 *
 * 参数说明：
 *     tree 二叉树的
 *     z 插入的结点
 */
private void insert(BSTree<T> bst, BSTNode<T> z) {
    int cmp;
    BSTNode<T> y = null;
    BSTNode<T> x = bst.mRoot;
```

```
    // 查找z的插入位置
    while (x != null) {
        y = x;
        cmp = z.key.compareTo(x.key);
        if (cmp < 0)
            x = x.left;
        else
            x = x.right;
    }

    z.parent = y;
    if (y==null)
        bst.mRoot = z;
    else {
        cmp = z.key.compareTo(y.key);
        if (cmp < 0)
            y.left = z;
        else
            y.right = z;
    }
}

/*
 * 新建结点(key)，并将其插入到二叉树中
 *
 * 参数说明：
 *     tree 二叉树的根结点
 *     key 插入结点的键值
 */
public void insert(T key) {
    BSTNode<T> z=new BSTNode<T>(key,null,null,null);

    // 如果新建结点失败，则返回。
    if (z != null)
        insert(this, z);
}
```

注：本文实现的二叉查找树是允许插入相同键值的节点的。

7. 删除

删除节点的代码

```
/*
 * 删除结点(z)，并返回被删除的结点
 *
 * 参数说明：
 *     bst 二叉树
 *     z 删除的结点
 */
private BSTNode<T> remove(BSTree<T> bst, BSTNode<T> z) {
    BSTNode<T> x=null;
    BSTNode<T> y=null;

    if ((z.left == null) || (z.right == null) )
        y = z;
    else
        y = successor(z);

    if (y.left != null)
        x = y.left;
    else
        x = y.right;

    if (x != null)
        x.parent = y.parent;

    if (y.parent == null)
        bst.mRoot = x;
    else if (y == y.parent.left)
        y.parent.left = x;
    else
        y.parent.right = x;

    if (y != z)
        z.key = y.key;

    return y;
}
```

```
/*
 * 删除结点(z)，并返回被删除的结点
 *
 * 参数说明：
 *     tree 二叉树的根结点
 *     z 删除的结点
 */
public void remove(T key) {
    BSTNode<T> z, node;

    if ((z = search(mRoot, key)) != null)
        if ( (node = remove(this, z)) != null)
            node = null;
}
```

8. 打印

打印二叉查找树的代码

```
/*
 * 打印"二叉查找树"
 *
 * key        -- 节点的键值
 * direction  --  0，表示该节点是根节点;
 *               -1，表示该节点是它的父结点的左孩子;
 *                1，表示该节点是它的父结点的右孩子。
 */
private void print(BSTNode<T> tree, T key, int direction) {

    if(tree != null) {

        if(direction==0)    // tree是根节点
            System.out.printf("%2d is root\n", tree.key);
        else                // tree是分支节点
            System.out.printf("%2d is %2d's %6s child\n", tree.k
ey, key, direction==1?"right" : "left");

        print(tree.left, tree.key, -1);
        print(tree.right,tree.key,  1);
    }
}

public void print() {
    if (mRoot != null)
        print(mRoot, mRoot.key, 0);
}
```

9. 销毁

销毁二叉查找树的代码

```
/*
 * 销毁二叉树
 */
private void destroy(BSTNode<T> tree) {
    if (tree==null)
        return ;

    if (tree.left != null)
        destroy(tree.left);
    if (tree.right != null)
        destroy(tree.right);

    tree=null;
}

public void clear() {
    destroy(mRoot);
    mRoot = null;
}
```

四、树的深度/广度优先遍历

树的深度优先遍历需要用到额外的数据结构--->栈；而广度优先遍历需要队列来辅助；这里以二叉树为例来实现。

```
import java.util.ArrayDeque;

public class BinaryTree {
    static class TreeNode{
        int value;
        TreeNode left;
        TreeNode right;

        public TreeNode(int value){
            this.value=value;
        }
    }
```

```
    TreeNode root;

    public BinaryTree(int[] array){
        root=makeBinaryTreeByArray(array,1);
    }

    /**
     * 采用递归的方式创建一颗二叉树
     * 传入的是二叉树的数组表示法
     * 构造后是二叉树的二叉链表表示法
     */
    public static TreeNode makeBinaryTreeByArray(int[] array,int
index){
        if(index<array.length){
            int value=array[index];
            if(value!=0){
                TreeNode t=new TreeNode(value);
                array[index]=0;
                t.left=makeBinaryTreeByArray(array,index*2);
                t.right=makeBinaryTreeByArray(array,index*2+1);
                return t;
            }
        }
        return null;
    }

    /**
     * 深度优先遍历，相当于先根遍历
     * 采用非递归实现
     * 需要辅助数据结构：栈
     */
    public void depthOrderTraversal(){
        if(root==null){
            System.out.println("empty tree");
            return;
        }
        ArrayDeque<TreeNode> stack=new ArrayDeque<TreeNode>();
        stack.push(root);
        while(stack.isEmpty()==false){
            TreeNode node=stack.pop();
```

```
            System.out.print(node.value+"    ");
            if(node.right!=null){
                stack.push(node.right);
            }
            if(node.left!=null){
                stack.push(node.left);
            }
        }
        System.out.print("\n");
    }

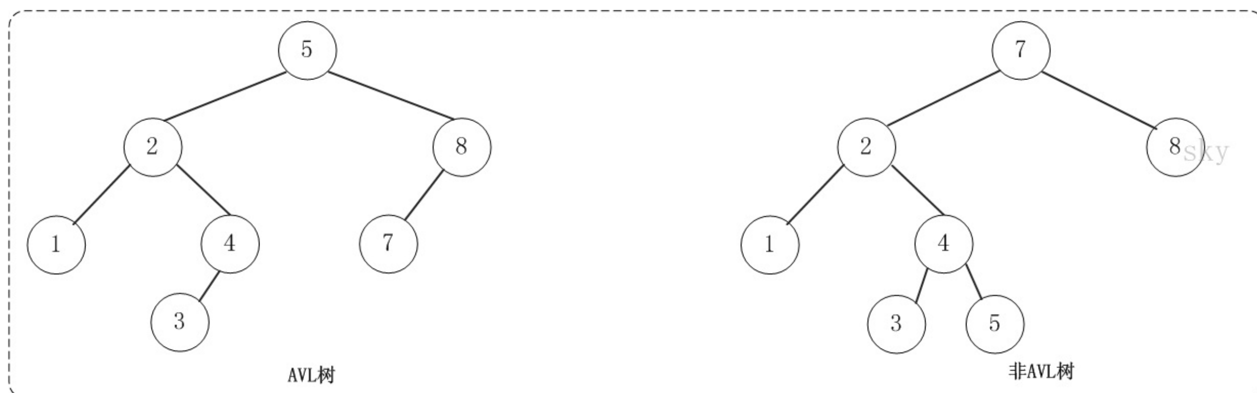
    /**
     * 广度优先遍历
     * 采用非递归实现
     * 需要辅助数据结构：队列
     */
    public void levelOrderTraversal(){
        if(root==null){
            System.out.println("empty tree");
            return;
        }
        ArrayDeque<TreeNode> queue=new ArrayDeque<TreeNode>();
        queue.add(root);
        while(queue.isEmpty()==false){
            TreeNode node=queue.remove();
            System.out.print(node.value+"    ");
            if(node.left!=null){
                queue.add(node.left);
            }
            if(node.right!=null){
                queue.add(node.right);
            }
        }
        System.out.print("\n");
    }

    /**
     *                  13
     *                 /  \
     *               65    5
```

```
     *              /  \    \
     *             97  25   37
     *            /    /\   /
     *           22   4 28 32
     */
    public static void main(String[] args) {
        int[] arr={0,13,65,5,97,25,0,37,22,0,4,28,0,0,32,0};
        BinaryTree tree=new BinaryTree(arr);
        tree.depthOrderTraversal();
        tree.levelOrderTraversal();
    }
}
```

一、AVL树

AVL树是高度平衡的而二叉树。它的特点是：AVL树中任何节点的两个子树的高度最大差别为1。



上面的两张图片，左边的是AVL树，它的任何节点的两个子树的高度差别都<=1；而右边的不是AVL树，因为7的两颗子树的高度相差为2(以2为根节点的树的高度是3，而以8为根节点的树的高度是1)。

二、红黑树

R-B Tree，全称是Red-Black Tree，又称为“红黑树”，它一种特殊的二叉查找树。红黑树的每个节点上都有存储位表示节点的颜色，可以是红(Red)或黑(Black)。

红黑树的特性:

（**1**）每个节点或者是黑色，或者是红色。

（**2**）根节点是黑色。

（**3**）每个叶子节点（**NIL**）是黑色。 **[**注意：这里叶子节点，是指为空**(NIL**或**NULL)**的叶子节点！**]**

（**4**）如果一个节点是红色的，则它的子节点必须是黑色的。

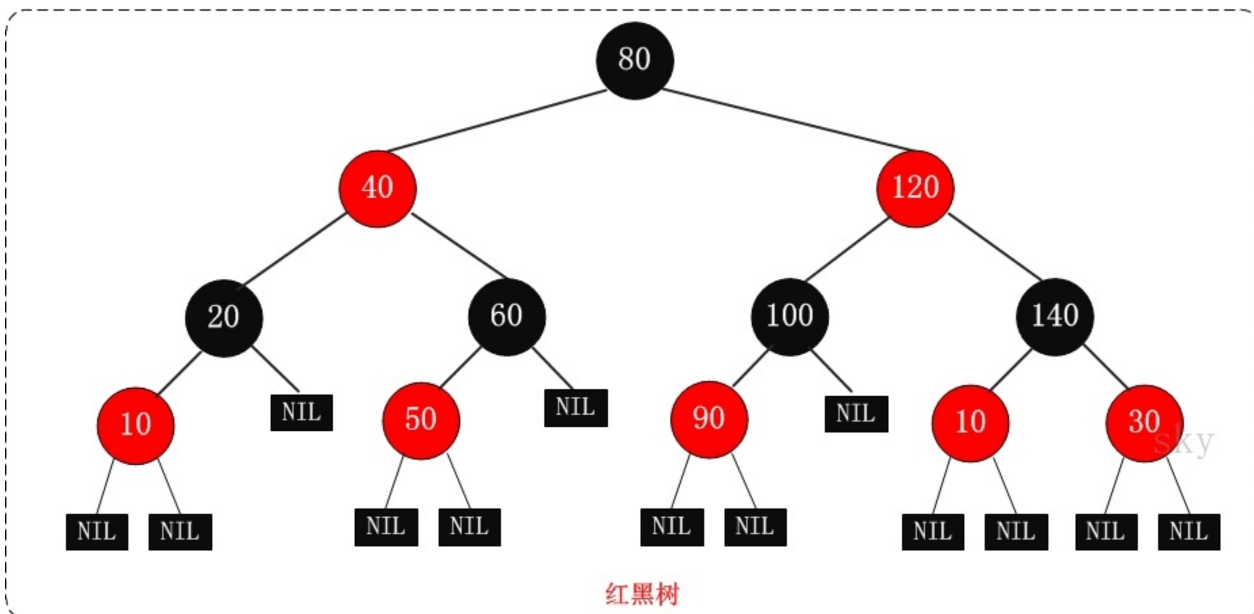
（**5**）从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。

注意：

(01) 特性(3)中的叶子节点，是只为空(NIL或null)的节点。

(02) 特性(5)，确保没有一条路径会比其他路径长出俩倍。因而，红黑树是相对是接近平衡的二叉树。

红黑树示意图如下：



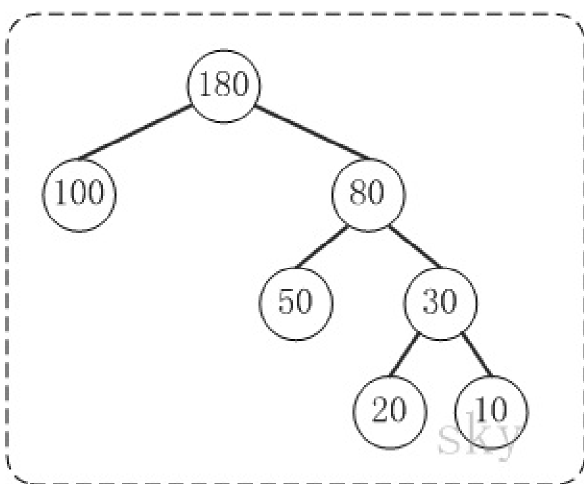
红黑树的应用

红黑树的应用比较广泛，主要是用它来存储有序的数据，它的时间复杂度是O(lgn)，效率非常之高。 例如，Java集合中的TreeMap和HashMap，都是通过红黑树去实现的。

三、哈夫曼树

Huffman Tree，中文名是哈夫曼树或霍夫曼树，它是最优二叉树。

定义：给定n个权值作为n个叶子结点，构造一棵二叉树，若树的带权路径长度达到最小，则这棵树被称为哈夫曼树。 这个定义里面涉及到了几个陌生的概念，下面就是一颗哈夫曼树，我们来看图解答。



(01) 路径和路径长度

> 定义：在一棵树中，从一个结点往下可以达到的孩子或孙子结点之间的通路，称为路径。通路中分支的数目称为路径长度。若规定根结点的层数为1，则从根结点到第L层结点的路径长度为L-1。 例子：100和80的路径长度是1，50和30的路径长度是2，20和10的路径长度是3。

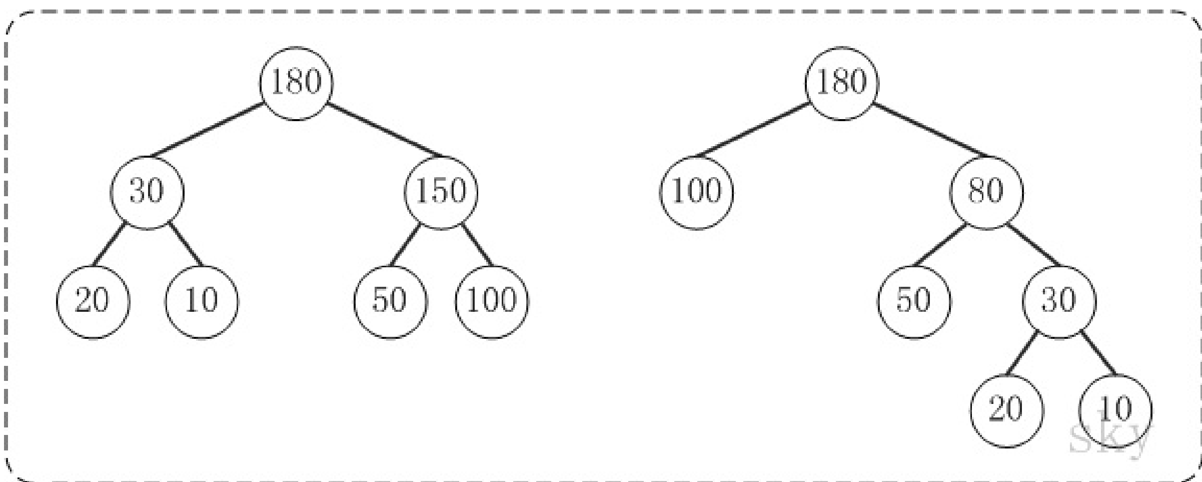
(02) 结点的权及带权路径长度

> 定义：若将树中结点赋给一个有着某种含义的数值，则这个数值称为该结点的权。结点的带权路径长度为：从根结点到该结点之间的路径长度与该结点的权的乘积。 例子：节点20的路径长度是3，它的带权路径长度= 路径长度x权 = 3 x 20 = 60。

(03) 树的带权路径长度

> 定义：树的带权路径长度规定为所有叶子结点的带权路径长度之和，记为WPL。 例子：示例中，树的WPL= 1x100 + 2x50 + 3x20 + 3x10 = 100 + 100 + 60 + 30 = 290。

比较下面两棵树



上面的两棵树都是以{10, 20, 50, 100}为叶子节点的树。

> 左边的树WPL=2x10 + 2x20 + 2x50 + 2x100 = 360
>
> 右边的树WPL=290

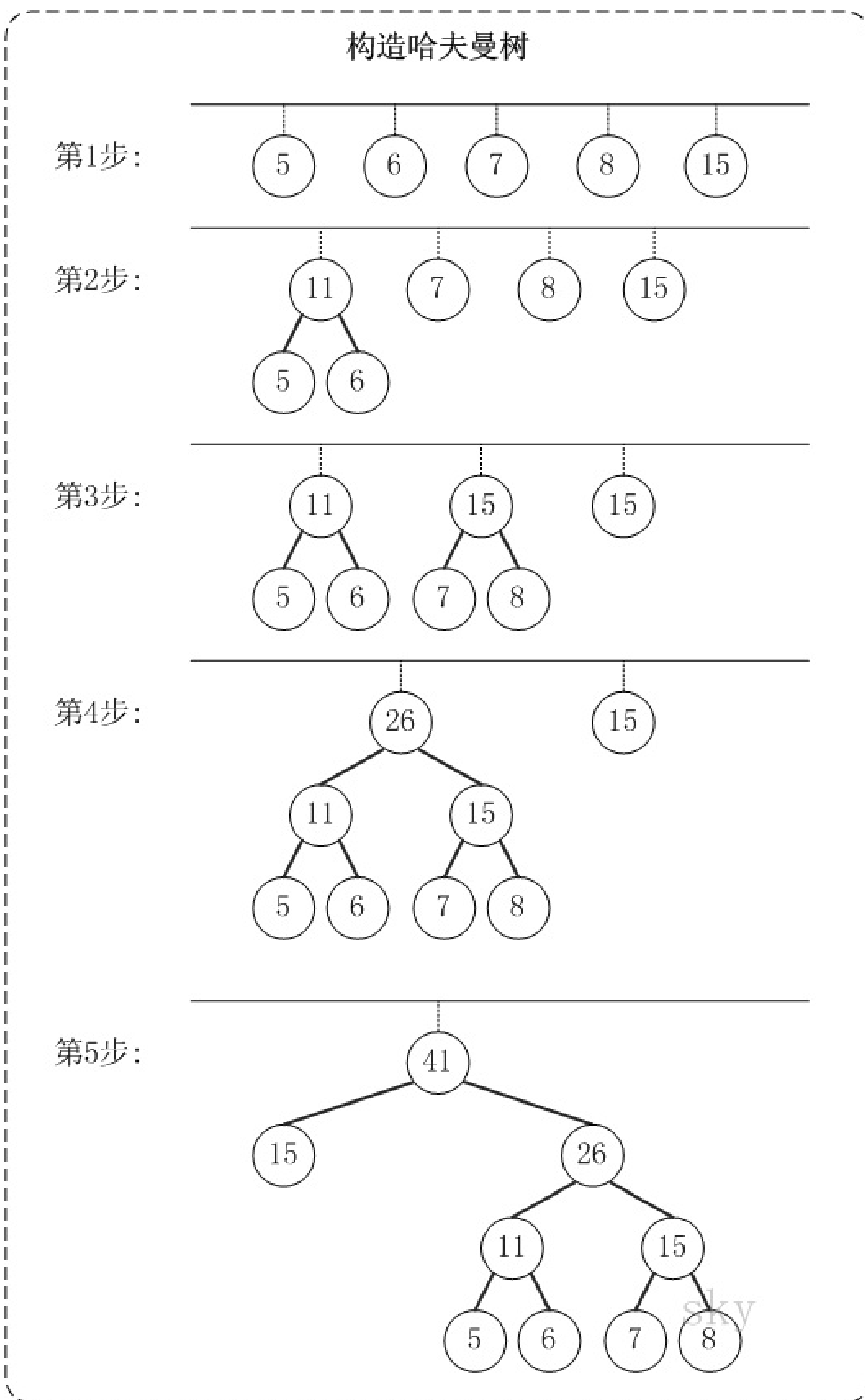
左边的树WPL > 右边的树的WPL。你也可以计算除上面两种示例之外的情况，但实际上右边的树就是{10,20,50,100}对应的哈夫曼树。至此，应该堆哈夫曼树的概念有了一定的了解了，下面看看如何去构造一棵哈夫曼树。

哈夫曼树的图文解析

假设有n个权值，则构造出的哈夫曼树有n个叶子结点。 n个权值分别设为 w1、w2、…、wn，哈夫曼树的构造规则为：

> **1**. 将w1、w2、…，wn看成是有n 棵树的森林(每棵树仅有一个结点)；
>
> **2**. 在森林中选出根结点的权值最小的两棵树进行合并，作为一棵新树的左、右子树，且新树的根结点权值为其左、右子树根结点权值之和；
>
> **3**. 从森林中删除选取的两棵树，并将新树加入森林；
>
> **4**. 重复(02)、(03)步，直到森林中只剩一棵树为止，该树即为所求得的哈夫曼树。

以{5,6,7,8,15}为例，来构造一棵哈夫曼树。



第1步：创建森林，森林包括5棵树，这5棵树的权值分别是5,6,7,8,15。

第2步：在森林中，选择根节点权值最小的两棵树(5和6)来进行合并，将它们作为一颗新树的左右孩子(谁左谁右无关紧要，这里，我们选择较小的作为左孩子)，并且新树的权值是左右孩子的权值之和。即，新树的权值是11。 然后，将"树5"和"树

6"从森林中删除，并将新的树(树11)添加到森林中。

第**3**步：在森林中，选择根节点权值最小的两棵树(7和8)来进行合并。得到的新树的权值是15。 然后，将"树7"和"树8"从森林中删除，并将新的树(树15)添加到森林中。

第**4**步：在森林中，选择根节点权值最小的两棵树(11和15)来进行合并。得到的新树的权值是26。 然后，将"树11"和"树15"从森林中删除，并将新的树(树26)添加到森林中。

第**5**步：在森林中，选择根节点权值最小的两棵树(15和26)来进行合并。得到的新树的权值是41。 然后，将"树15"和"树26"从森林中删除，并将新的树(树41)添加到森林中。

此时，森林中只有一棵树(树41)。这棵树就是我们需要的哈夫曼树！

一、并查集的介绍

并查集（Union/Find）从名字可以看出，主要涉及两种基本操作:合并和查找。这说明，初始时并查集中的元素是不相交的，经过一系列的基本操作(Union)，最终合并成一个大的集合。

而在某次合并之后，有一种合理的需求：某两个元素是否已经处在同一个集合中了？因此就需要Find操作。

并查集是一种 不相交集合 的数据结构，设有一个动态集合S={s1，s2，s3，.....sn}，每个集合通过一个代表来标识，该代表中集合中的某个元素。

比如，若某个元素 x 是否在集合 s1 中(Find操作)，返回集合 s1 的代表元素即可。这样，判断两个元素是否在同一个集合中也是很方便的，只要看find(x) 和 find(y) 是否返回同一个代表即可。

为什么是动态集合S呢？因为随着Union操作，动态集合S中的子集合个数越来越少。

数据结构的基本操作决定了它的应用范围，对并查集而言，一个简单的应用就是判断无向图的连通分量个数，或者判断无向图中任何两个顶点是否连通。

二、并查集的存储结构及实现分析

①存储结构

并查集(大S)由若干子集合si构成，并查集的逻辑结构就是一个森林。si表示森林中的一棵子树。一般以子树的根作为该子树的代表。

而对于并查集的存储结构，可用一维数组和链表来实现。这里主要介绍一维数组的实现。

根据前面介绍的基本操作再加上存储结构，并查集类的实现架构如下：

```
public class DisjSets {
    private int[] s;
    private int count;//记录并查集中子集合的个数(子树的个数)

       public DisjSets(int numElements) {
       //构造函数，负责初始化并查集
       }

        public void unionByHeight(int root1, int root2){
        //union操作
       }

       public int find(int x){
       //find 操作
      }
}
```

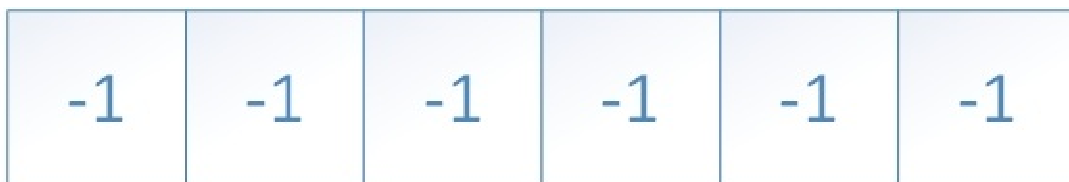
由于Find操作需要找到该子集合的代表元素，而代表元素是树根，因此需要保存树中结点的父亲，对于每一个结点，如果知道了父亲，沿着父结点链就可以最终找到树根。

为了简单起见，假设一维数组s中的每个元素 s[i] 表示该元素 i 的父亲。这里有两个需要注意的地方：①我们用一维数组来存储并查集，数组的元素s[i]表示的是结点的父亲的位置。②数组元素的下标 i 则是结点的标识。如：s[5]=4，节点5 在数组的第4号位置处。

假设有并查集中6个元素，初始时，所有的元素都相互独立，处在不同的集合中：



对应的一维数组初始化如下：



-1	-1	-1	-1	-1	-1

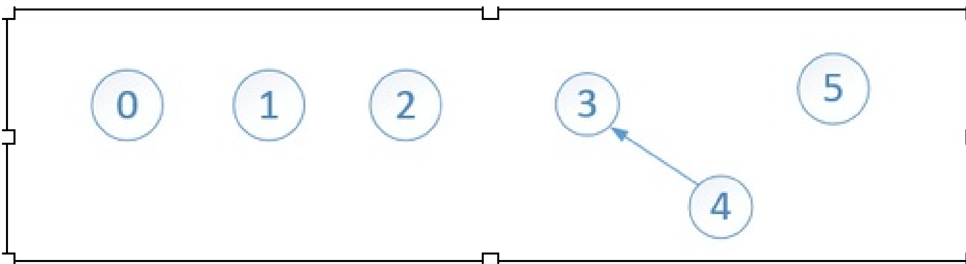
因为，初始时每个元素代表一个集合，该元素本身就是树根。树根的父结点用 -1 来表示。代码实现如下：

```
public DisjSets(int numElements) {
        s = new int[numElements];
        count = numElements;
        //初始化并查集,相当于新建了s.length 个互不相交的集合
        for(int i = 0; i < s.length; i++)
            s[i] = -1;//s[i]存储的是高度(秩)信息
    }
```

②基本操作实现

Union操作就是将两个不相交的子集合合并成一个大集合。简单的Union操作是非常容易实现的，因为只需要把一棵子树的根结点指向另一棵子树即可完成合并。

比如合并 节点3 和节点4：



这里的合并很随意，把任意一棵子树的结点指向另一棵子树结点就完成了合并。

```
public void union(int root1, int root2){
  s[root2] = root1;//将root1作为root2的新树根
}
```

但是，这只是一个简单的情况，如果待合并的两棵子树很大，而且高度不一样时，如何使得合并操作生成的新的子树的高度最小？因为高度越小的子树Find操作越快。

后面会介绍一种更好的合并策略，以支持Quick Union/Find。

Find操作就是查找某个元素所在的集合，返回该集合的代表元素。在union(3,4) 和union(1,2)后，并查集如下：



此时的一维数组如下：

-1	-1	1	-1	3	-1

此时一共有4个子集合。第一个集合的代表元素为0，第二个集合的代表元素为1，第三个集合的代表元素为3，第四个集合的代表元素为5，故：

find(2)返回1，find(0)返回0。因为 结点3 和 结点4 在同一个集合内，find(4)返回3，find(3)返回3。

```
public int find(int x){
 if(s[x] < 0)
   return x;
 else
   return find(s[x]);
}
```

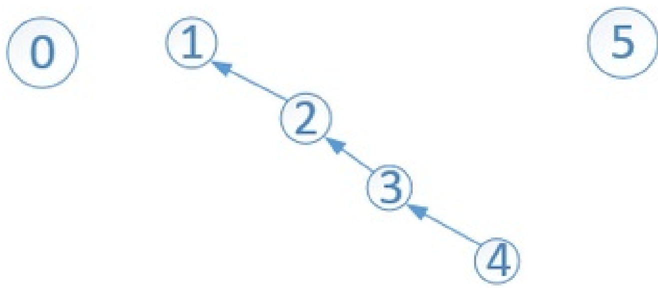
这里find(int x)返回的是最里层递归执行后，得到的值。由于只有树根的父结点位置小于0，故返回的是树根结点的标识。

（数组中索引 i 处的元素 s[i] 小于0，表示 结点i 是根结点.....）

三、Union/Find的改进----Quick Union/Find

上面介绍的Union操作很随意：任选一棵子树，将另一棵子树的根指向它即完成了合并。如果一直按照上述方式合并，很可能产生一棵非常不平衡的子树。

比如在上面的基础上union(2,3)后



树越来越高了，此时会影响到Find操作的效率。比如，find(4)时，会一直沿着父结点遍历直到根，4-->3-->2-->1

这里引入一种新的合并策略，这是一种启发式策略，称之为按秩合并：将秩小的子树的根指向秩大的子树的根。

秩的定义：对每个结点，用秩表示结点高度的一个上界。为什么是上界？

因为路径压缩不完全与按高度求并兼容。路径压缩会改变树的高度，这样在Union操作之前，我们就无法获得子树的高度的精确值，因此就不计算高度的精确值，而是存储每棵树的高度的估计值，这个值称之为秩。

说了这么多，按秩求并就是在合并之前，先判断下哪棵子树更高，让矮的子树的根指向高的子树的根。

除了按高度求并之外，还可以按大小求并，即先判断下哪棵子树含有的结点数目多，让较小的子树的根指向较大的子树的根。

对于按高度求并，需要解释下数组中存储的元素：是高度的负值再减去1。这样，初始时，所有元素都是-1，而树根节点的高度为0，s[i]=-1。

按高度求并的代码如下：

```
/**
     *
     * @param root1 并查集中以root1为代表的某个子集
     * @param roo2 并查集中以root2为代表的某个子集
     * 按高度(秩)合并以root1 和 root2为代表的两个集合
     */
    public void unionByHeight(int root1, int root2){
        if(find(root1) == find(root2))
            return;//root1 与 root2已经连通了

        if(s[root2] < s[root1])//root2 is deeper
            s[root1] = root2;
        else{
            if(s[root1] == s[root2])//root1 and root2 is the sam
e deeper
                s[root1]--;//将root1的高度加1
            s[root2] = root1;//将root2的根(指向)更新为root1
        }

        count--;//每union一次,子树数目减1
    }
```

使用了路径压缩的**Find**的操作

上面程序代码find方法只是简单地把待查找的元素所在的根返回。路径压缩是指，在find操作进行时，使**find**查找路径中的顶点**(**的父亲**)**都直接指向为树根（这很明显地改变了子树的高度）

如何使find查找路径中经过的每个顶点都直接指向树根呢？只需要小小改动一下就可以了，这里用到了非常神奇的递归。修改后的find代码如下：

```
public int find(int x){
        if(s[x] < 0)//s[x]为负数时,说明 x 为该子集合的代表(也即树根),且s[x]的值表示树的高度
            return x;
        else
            return s[x] = find(s[x]);//使用了路径压缩,让查找路径上的所有顶点都指向了树根(代表节点)
            //return find(s[x]); 没有使用 路径压缩
    }
```

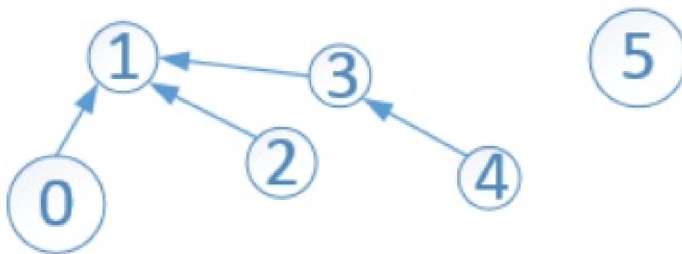
因为递归最终得到的返回值是根元素。第5行将根元素直接赋值给s[x]，s[x]在每次递归过程中相当于结点x的父结点指针。

关于路径压缩对按"秩"求并的兼容性问题

上面的unionByHeight(int , int)是按照两棵树的高度来进行合并的。但是find操作中的路径压缩会对树的高度产生影响。使用了路径压缩后，树的高度变化了，但是数组并没有更新这个变化。因为无法更新！！（我们没有在Find操作中去计算原来的树的高度，然后再计算新的树的高度，这样不现实，复杂度太大了）

举个例子：

依次高度unionByHeight(3, 4)、unionByHeight(1, 3)、unionByHeight(1, 0)后，并查集如下：



此时，数组中的元素如下：

1	-3	1	1	3	-1

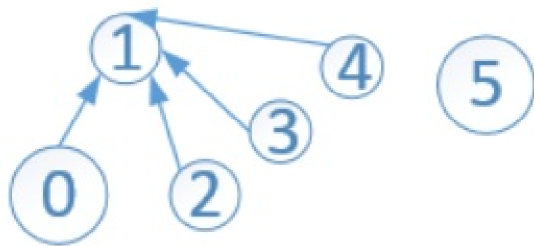
可以看出，此时只有两棵子树，一棵根结点为1，另一棵只有一个结点5。结点1的s[1]=-3，它所表示是该子树的高度为2，如果此时执行find(4)，会改变这棵树的高度！但是，数组s中存储的根的高度却没有更新，只会更新查找路径上的顶点的高度。执行完find(4)后，变成：

1	-3	1	1	1	-1

查找路径为 **4-->3-->1**，**find(4)**使得查找路径上的所有顶点的父结点指向了根。如，将结点**4** 指向了根。但是没有根结点的高度（没有影响树根的秩），因为**s[1]**的值仍为**-3**

-3表示的高度为**2**，但是树的高度实际上已经变成了**1**

执行find(4)之后，树实际上是这样的：



（关于路径压缩对按秩合并有影响，我一直有个疑问，希望有大神指点啊）。。。。

路径压缩改变了子树的高度，而这个高度是按秩求的依据。，而且当高度改变之后，我们是无法更新这个变化了的高度的。那这会不会影响按秩求并的正确性？或者说使按秩求并达不到减小新生成的子树的高度的效果？

四、并查集的应用

并查集数据结构非常简单，基本操作也很简单。但是用途感觉很大。比如，求解无向图中连通分量的个数，生成迷宫......

这些应用本质上就是：初始时都是一个个不连通的对象，经过一步步处理，变成连通的了。。。。。

如迷宫，初始时，起点和终点不连通，随机地打开起点到终点路径上的一个方向，直至起点和终点连通了，就生成了一个迷宫。

如，无向图的连通分量个数，初始时，将无向图中各个顶点视为不连通的子集合，对图中每一条边，相当于union这条边对应的两个顶点分别所在的集合，直至所有的边都处理完后，还剩下的集合的个数即为连通分量的个数。

五、完整代码

```
public class DisjSets {
    private int[] s;
    private int count;//记录并查集中子集合的个数(子树的个数)


    public DisjSets(int numElements) {
        s = new int[numElements];
        count = numElements;
        //初始化并查集,相当于新建了s.length 个互不相交的集合
        for(int i = 0; i < s.length; i++)
            s[i] = -1;//s[i]存储的是高度(秩)信息
    }

    /**
     *
     * @param root1 并查集中以root1为代表的某个子集
     * @param roo2 并查集中以root2为代表的某个子集
     * 按高度(秩)合并以root1 和 root2为代表的两个集合
     */
    public void unionByHeight(int root1, int root2){
        if(find(root1) == find(root2))
            return;//root1 与 root2已经连通了

        if(s[root2] < s[root1])//root2 is deeper
            s[root1] = root2;
        else{
            if(s[root1] == s[root2])//root1 and root2 is the sam
e deeper
                s[root1]--;//将root1的高度加1
            s[root2] = root1;//将root2的根(指向)更新为root1
        }
```

```
        count--;//每union一次,子树数目减1
    }

    public void union(int root1, int root2){
        s[root2] = root1;//将root1作为root2的新树根
    }


    public void unionBySize(int root1, int root2){

        if(find(root1) == find(root2))
            return;//root1 与 root2已经连通了

        if(s[root2] < s[root1])//root2 is deeper
            s[root1] = root2;
        else{
            if(s[root1] == s[root2])//root1 and root2 is the sam
e deeper
                s[root1]--;//将root1的高度加1
            s[root2] = root1;//将root2的根(指向)更新为root1
        }

        count--;//每union一次,子树数目减1
    }


    public int find(int x){
        if(s[x] < 0)//s[x]为负数时,说明 x 为该子集合的代表(也即树根),
且s[x]的值表示树的高度
            return x;
        else
            return s[x] = find(s[x]);//使用了路径压缩,让查找路径上的
所有顶点都指向了树根(代表节点)
            //return find(s[x]); 没有使用 路径压缩
    }

    public int find0(int x){
        if(s[x] < 0)
            return x;
```

```
        else
            return find0(s[x]);
    }


    public int size(){
        return count;
    }
}
```

一、概念

1. Binary Tree（二叉树）：二叉树的每个节点最多有两个子节点
2. Binary Search Tree（二叉搜索树）：二叉搜索树每个节点只存储一个键值，并且左子树（如果有）所有节点的值都要小于根节点的值，右子树（如果有）所有节点的值都要大于根节点的值。
3. B-Tree（Balanced Tree）：也就是今天要说的B-树，这里的-不是minus的意思，而是作为连接符的横杠，而我们也经常把B-树直接翻译为B树，所以B树与B-树通常是指一个概念，B代表的是Balance，而不是Binary。而B+树和B*树则是B-树的基础上正对不同场景的优化版本，将会在后文中有所介绍。

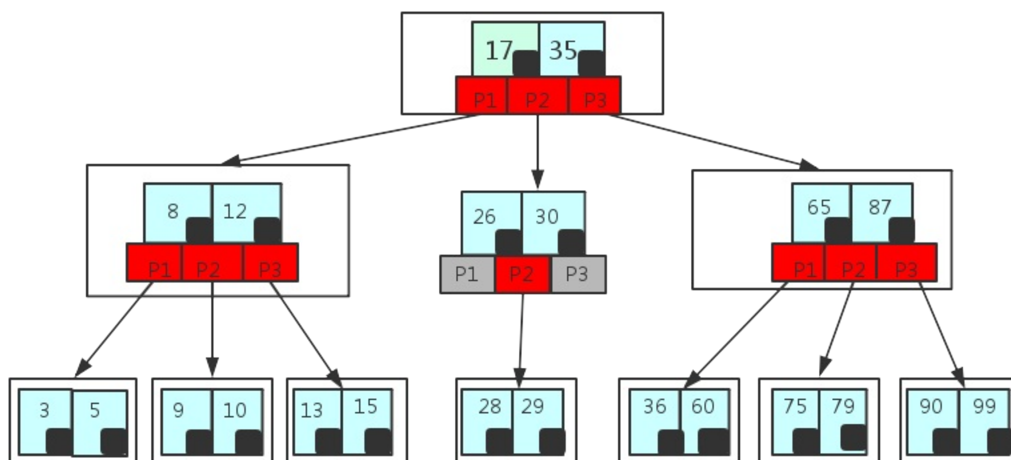
在大规模数据存储中，二叉查找树的深度会过大，当内存无法存储所有节点数据时，需要读取磁盘，进行IO操作，从而树的高度越高，I/O操作次数越多，效率也就越低。所以诸如之前所讲的红黑树，AVL树 因为树的高度太高而不适合这种需要大量IO操作的查询。所以，B树通过多叉的实现来降低树的高度，从而减少IO操作的次数。

二、B树（B-树）

为方便描述，下面一律用B数这个名称。B树是一种多路平衡搜索树（非二叉），若其是M路，则：

1. 任意非叶子节点最多可以有M个子女，且M>2；
2. 根节点的子女数为[2,M]；
3. 除了根节点以外的非叶子节点的子女数目为M/2（取上整）个到M个；
4. 每个节点存放至少M/2-1（取上整）和至多M-1个键值（至少两个）；
5. 非叶子节点的关键字个数=指向子女的指针个数-1；
6. 非叶子节点的关键字K[1],K[2],…,K[M-1]且有K[i]<K[i+1]；
7. 非叶子节点的指针P[1],P[2],…,P[M]；其中P[1]指向关键字小于K[1]的子树，P[M]指向关键字大于K[M-1]的子树，其他P[i]指向关键字属于(K[i-1],K[i])的子树；
8. 所有叶子节点都位于同一层。

B树与二叉搜索树的最大区别在于其每个节点可以存不止一个键值，并且其子女不止两个，不过还是需要满足键值数=子女数-1。因此，对于相同数量的键值，B树比二叉搜索树要更加矮一些，特别是当M较大时，树高会更低。



B树

上图中是一个简单的B树，在实际应用中，M可以取到很大，比如大于1000。一般情况下M的取值会使得每个磁盘盘块可以正好存放一个B数节点。上图中的35节点，35是一个key（或者说是索引，比如磁盘文件的文件名），而小黑块则代表的是该key所指向的内容在磁盘中实际的存储位置，是一个指针（比如35这个文件在硬盘中的位置）。

B树的搜索

B树的搜索与二叉搜索树类似，只不过需要在节点内部进行一次搜索查找。从根结点开始，对结点内的关键字（有序）序列进行二分查找，如果命中则结束，否则进入查询关键字所属范围的儿子结点；重复，直到所对应的儿子指针为空，或已经是叶子结点；

B树的插入

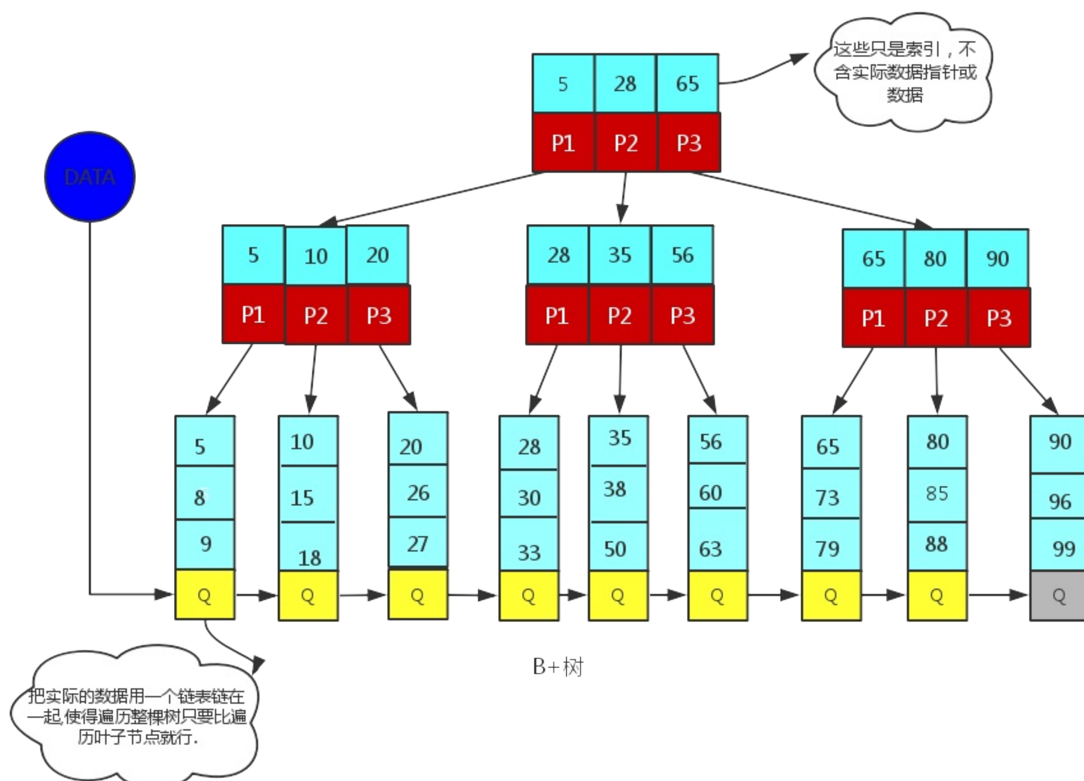
B树的插入首先查找插入所在的节点，若该节点未满，插入即可，若该节点以及满了，则需要将该节点分裂，并将该节点的中间的元素移动到父节点上，若父节点未满，则结束，若父节点也满了，则需要继续分裂父节点，如此不断向上，直到根节点，如果根节点也满了，则分裂根节点，从而树的高度+1。

下面是B树插入的一个演示动画，往B树中一次插入的元素为6 10 4 14 5 11 15 3 2 12 1 7 8 8 6 3 6 21 5 15 15 6 32 23 45 65 7 8 6 5 4。

B树的删除

B树的删除首先要找到删除的节点，并删除节点中的元素，如果删除的元素有左右孩子，则上移左孩子最右节点或右孩子最左节点到父节点，若没有左右孩子，则直接删除。删除后，若某节点中元素数目不符合B树要求（小于M/2-1取上整），则需要看起相邻的兄弟节点是否有多余的元素，若有，则可以向父节点借一个元素，然后将最丰满的相邻兄弟结点中上移最后或最前一个元素到父节点中（有点类似于左旋）。若其相邻兄弟节点没有多余的元素，则与其兄弟节点合并成一个节点，此时也需要将父节点中的一个元素一起合并。

三、B+树



B+树

B+树是B树的一个变种，其也是一种多路平衡搜索树，其与B树的主要区别是：

1. 非叶子节点的指针数量与关键字数量相等；
2. 非叶子节点的子树指针P[i]，指向关键字值属于[K[i],K[i+1]）的子树（B树是开区间，B+树是左闭右开，也就是说B树不允许关键字重复，而B+树允许）；
3. 所有关键字都在叶子节点出现，所有的叶子节点增加了一个链指针（稠密索引，且链表中的关键字切好是有序的）；
4. 非叶子节点相当于是叶子节点的索引（稀疏索引），叶子节点相当于是存储数据的数据层。

B+树主要是应文件系统所需而产生的。文件系统中，文件的目录是一级一级索引，只有最底层的叶子节点（文件）保存数据。非叶子节点只保存索引，不保存实际的数据，数据都保存在叶子节点中，所有的非叶子节点都可以看成是索引部分。

非叶子节点（比如[5，28，65]）只是一个key（索引，实际的数据在叶子节点上，对应于叶子节点[5,8,9]中的5，[28,30,33]中的28，[65,73,79]中的65才是真正的数据或指向真实数据的指针）。

B+树的搜索

B+的搜索与B树也是基本相同的。唯一的区别是B+树只有达到叶子结点才命中，因为只有叶节点中存放着真实数据或真实数据的指正，而B树可以在非叶子结点命中，其性能也等价于在元素全集做一次二分查找。

B+树的插入

B+树的插入与B树类似，如果节点中有多余的空间放入元素，则直接插入即可。如果节点本来就已经满了，则将其分裂为两个节点，并将其中间元素的索引放入到父节点中，在这里如果是叶子节点的话，是拷贝中间元素的索引到父节点中（因为叶子节点需要包含所有的元素），而如果是非叶子节点，则是上移节点的中间元素到父节点中。

下面是B+树插入的一个演示动画：

B+树的删除

在叶节点中删除元素，如果节点还满足B+树的要求，则okay。如果元素个数过少，并且其邻近兄弟节点有多余的元素，则从邻近兄弟节点中借一个元素，并修改父节点中的索引使其满足新的划分。如果其邻近兄弟节点也没有多余的元素，则将其和邻近兄弟节点合并，并且我们需要修改其父节点的索引以满足新的划分。并且如果父节点的索引元素太少不满足要求，则需要继续看起兄弟节点是否多余，如果没有多余则还需要与兄弟节点合并，如此不断向上，直到根节点。如果根节点中元素也被删除，则把根节点删除，并由合并来的节点作为新的根节点，树的高度减1。

四、B+树与B树的比较

B+树的非叶子节点并没有指向关键字具体信息的指针，因此其内部节点相对B树更小，如果把所有同一内部节点的关键字存放在同一盘块中，盘块所能容纳的关键字数量也越多，具有更好的空间局部性，一次性读入内存的需要查找的关键字也越多，相对的IO读写次数也就降低了。

另外对于B+树来说，因为非叶子节点只是叶子节点中关键字的索引，所以任何关键字的查找都必须走一条从根节点到叶子节点的路，所有关键字查询的路径长度相同。而若经常访问的元素离根节点很近，则B树访问更迅速，因为其不一定要到叶子节点。

数据库索引采用B+树的主要原因是B树在提高了IO性能的同时并没有解决元素遍历效率低下的问题，而也正是为了解决该问题，B+树应运而生。因为叶子节点中增加了一个链指针，B+树只需要取遍历叶子节点可以实现整棵树的遍历。而且数据库中基于范围的查询是非常频繁的，B树对基于范围的查询效率太低。

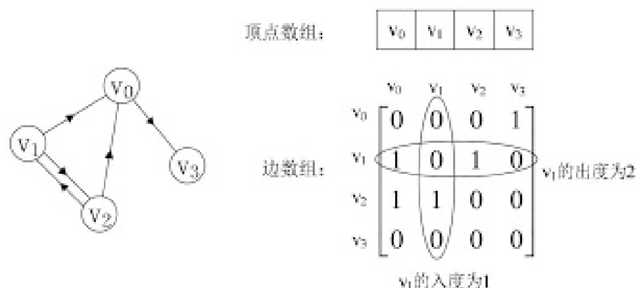
五、B*树

B*树又是B+树的变种，其与B+树的区别有：

1. B*树在B+树的非根和非叶子节点再增加指向兄弟节点的指针
2. B树规定非叶子节点的键值个数至少为*(2/3)*M，这样每个节点的使用率就从B+树的1/2上升到2/3，所以空间使用率更高。

B树的分裂：当一个结点满时，如果它的下一个兄弟结点未满，那么将一部分数据移到兄弟结点中，再在原结点插入关键字，最后修改父结点中兄弟结点的关键字（因为兄弟结点的关键字范围改变了）；如果兄弟也满了，则在原结点与兄弟结点之间增加新结点，并各复制*1/3*的数据到新结点，最后在父结点增加新结点的指针；*B*树分配新结点的概率比B+树要低，空间使用率更高；

一、前言



图作为数据结构中最复杂的一种结构，涉及到一些著名的算法，如dijkstra,flyod等。

此外，图的一些特性也值得我们学习，是笔试和面试中会高频考察的知识点。

本部分内容不仅介绍了图的基础概念及特性，同时介绍了图涉及的一些算法的Java实现。

二、目录

一、图的基本概念

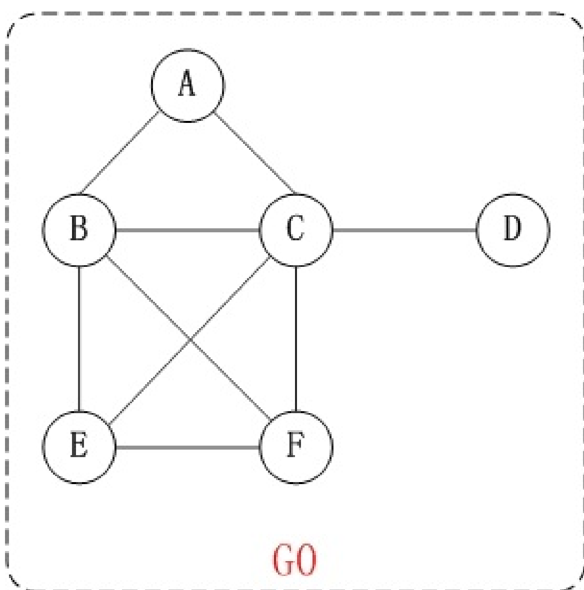
1. 图的定义

定义：图(graph)是由一些点(vertex)和这些点之间的连线(edge)所组成的；其中，点通常被成为"顶点(vertex)"，而点与点之间的连线则被成为"边或弧"(edege)。通常记为，G=(V,E)。

2. 图的种类

根据边是否有方向，将图可以划分为：无向图和有向图。

2.1 无向图

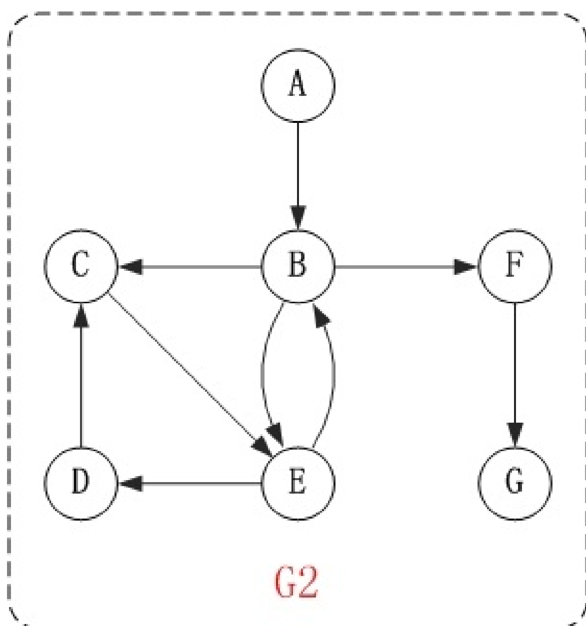


上面的图G0是无向图，无向图的所有的边都是不区分方向的。G0=(V1,{E1})。其中，

(01) V1={A,B,C,D,E,F}。 V1表示由"A,B,C,D,E,F"几个顶点组成的集合。

(02) E1={(A,B),(A,C),(B,C),(B,E),(B,F),(C,F), (C,D),(E,F),(C,E)}。 E1是由边(A,B),边(A,C)...等等组成的集合。其中，(A,C)表示由顶点A和顶点C连接成的边。

2.2 有向图



上面的图G2是有向图。和无向图不同，有向图的所有的边都是有方向的！ G2=(V2,{A2})。其中，

(01) V2={A,C,B,F,D,E,G}。 V2表示由"A,B,C,D,E,F,G"几个顶点组成的集合。

(02) A2={,,,,,,,,}。 E1是由矢量,矢量...等等组成的集合。其中，矢量<A,B)表示由"顶点A"指向"顶点B"的有向边。

3. 邻接点和度

3.1 邻接点

一条边上的两个顶点叫做邻接点。

例如，上面无向图*G0*中的顶点*A*和顶点*C*就是邻接点。

在有向图中，除了邻接点之外；还有"入边"和"出边"的概念。

顶点的入边，是指以该顶点为终点的边。而顶点的出边，则是指以该顶点为起点的边。

例如，上面有向图*G2*中的*B*和*E*是邻接点；是*B*的出边，还是*E*的入边。

3.2 度

在无向图中，某个顶点的度是邻接到该顶点的边(或弧)的数目。

例如，上面无向图*G0*中顶点*A*的度是*2*。

在有向图中，度还有"入度"和"出度"之分。

某个顶点的入度，是指以该顶点为终点的边的数目。而顶点的出度，则是指以该顶点为起点的边的数目。 顶点的度=入度+出度。

例如，上面有向图*G2*中，顶点*B*的入度是*2*，出度是*3*；顶点*B*的度=*2+3=5*。

4. 路径和回路

路径：如果顶点(Vm)到顶点(Vn)之间存在一个顶点序列。则表示Vm到Vn是一条路径。

路径长度：路径中"边的数量"。

简单路径：若一条路径上顶点不重复出现，则是简单路径。

回路：若路径的第一个顶点和最后一个顶点相同，则是回路。

简单回路：第一个顶点和最后一个顶点相同，其它各顶点都不重复的回路则是简单回路。

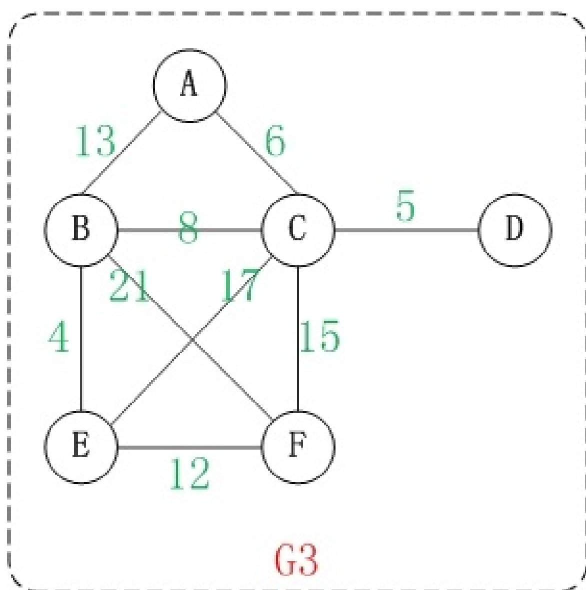
5. 连通图和连通分量

连通图：对无向图而言，任意两个顶点之间都存在一条无向路径，则称该无向图为连通图。 对有向图而言，若图中任意两个顶点之间都存在一条有向路径，则称该有向图为强连通图。

连通分量：非连通图中的各个连通子图称为该图的连通分量。

6. 权

在学习"哈夫曼树"的时候，了解过"权"的概念。图中权的概念与此类似。



上面就是一个带权的图。

二、图的存储结构

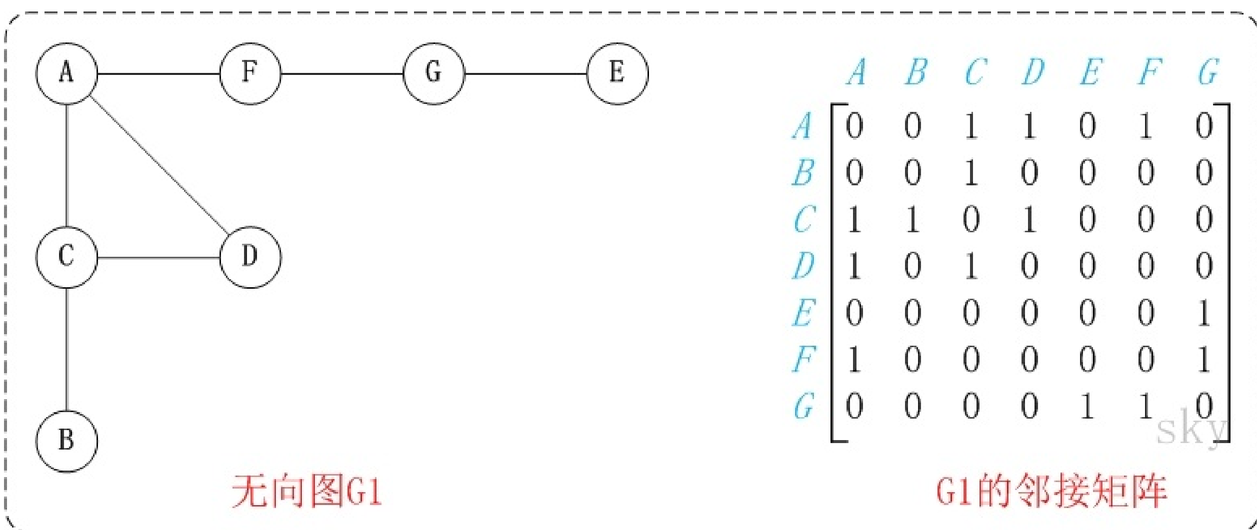
上面了解了"图的基本概念"，下面开始介绍图的存储结构。图的存储结构，常用的是"邻接矩阵"和"邻接表"。

1. 邻接矩阵

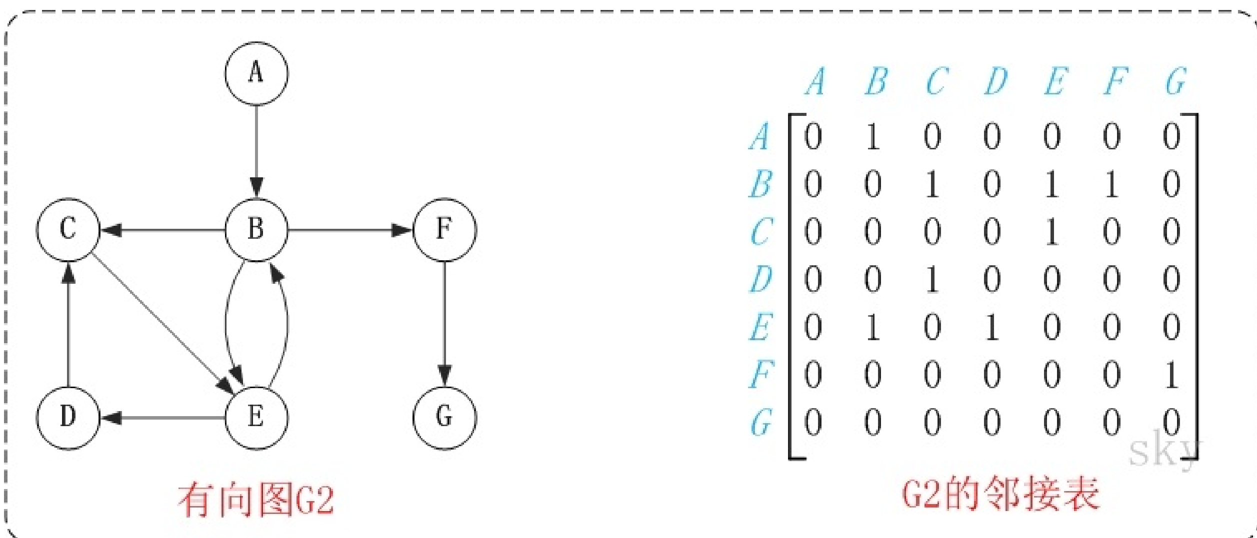
邻接矩阵是指用矩阵来表示图。它是采用矩阵来描述图中顶点之间的关系(及弧或边的权)。 假设图中顶点数为n，则邻接矩阵定义为：

$$A[i][j]=\begin{cases} 1 & \text{(若Vi和Vj之间有弧或边存在)} \\ 0 & \text{(Vi和Vj之间没有弧或边存在)} \end{cases}$$

下面通过示意图来进行解释。



图中的G1是无向图和它对应的邻接矩阵。

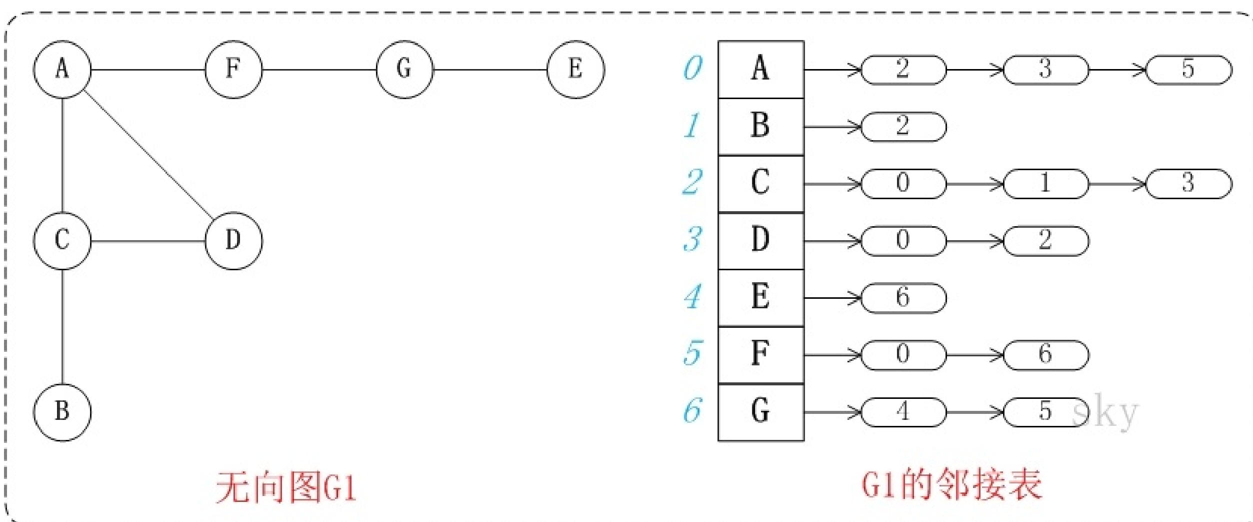


图中的G2是无向图和它对应的邻接矩阵。

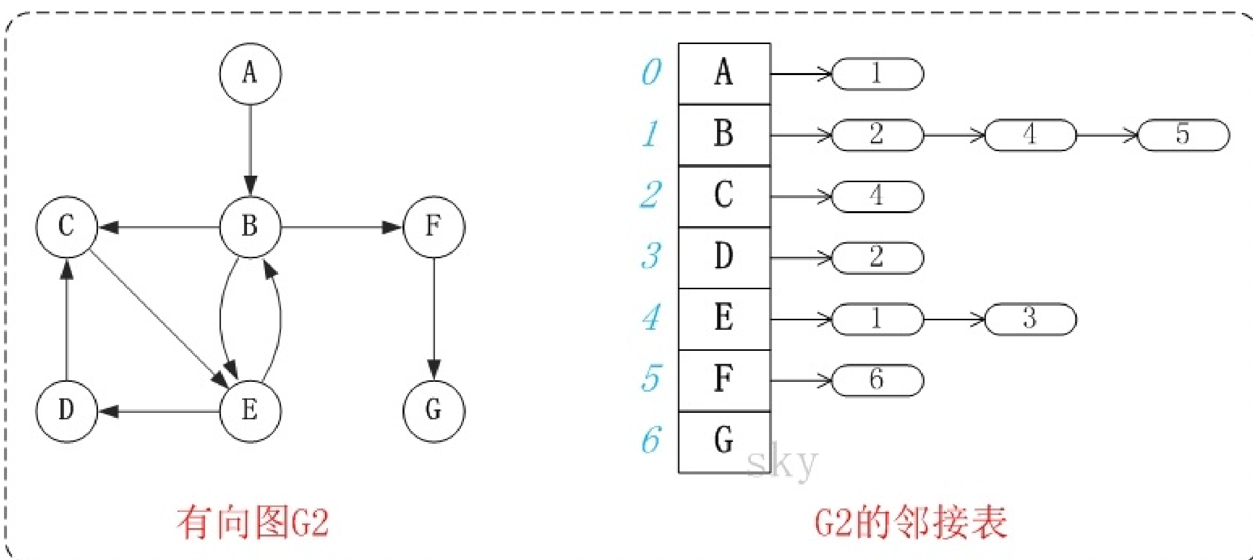
通常采用两个数组来实现邻接矩阵：一个一维数组用来保存顶点信息，一个二维数组来用保存边的信息。 邻接矩阵的缺点就是比较耗费空间。

2. 邻接表

邻接表是图的一种链式存储表示方法。它是改进后的"邻接矩阵"，它的缺点是不方便判断两个顶点之间是否有边，但是相对邻接矩阵来说更省空间。



图中的G1是无向图和它对应的邻接矩阵。



图中的G2是有向图和它对应的邻接矩阵。

三、图的深度/广度优先遍历

1. 深度优先搜索介绍

图的深度优先搜索(Depth First Search)，和树的先序遍历比较类似。

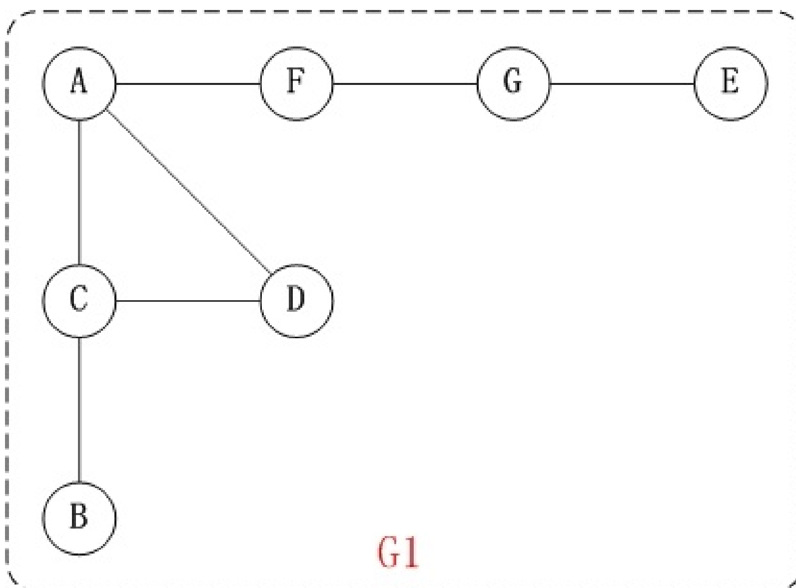
它的思想：假设初始状态是图中所有顶点均未被访问，则从某个顶点v出发，首先访问该顶点，然后依次从它的各个未被访问的邻接点出发深度优先搜索遍历图，直至图中所有和v有路径相通的顶点都被访问到。 若此时尚有其他顶点未被访问到，则另选一个未被访问的顶点作起始点，重复上述过程，直至图中所有顶点都被访问到为止。

显然，深度优先搜索是一个递归的过程。

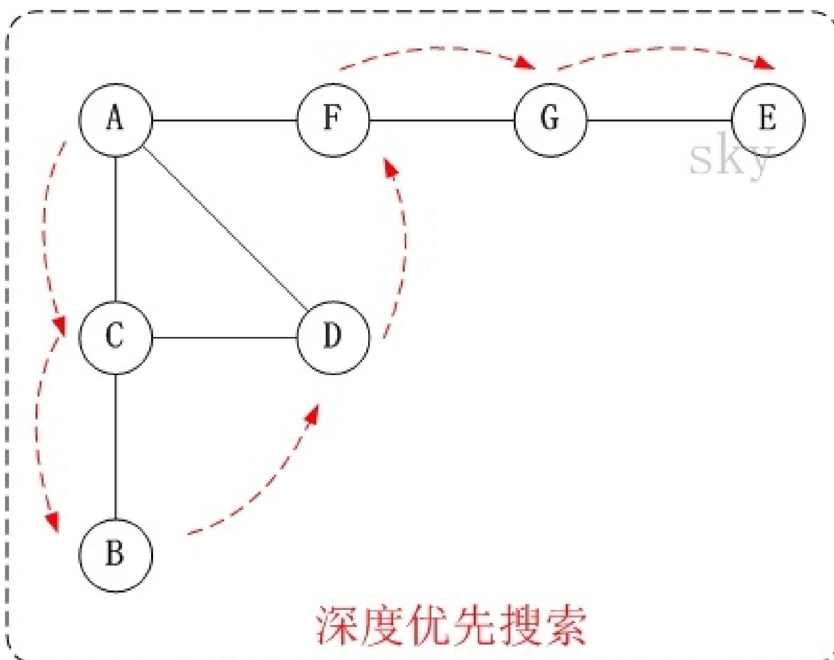
2. 深度优先搜索图解

2.1 无向图的深度优先搜索

下面以"无向图"为例，来对深度优先搜索进行演示。



对上面的图G1进行深度优先遍历，从顶点A开始。



第**1**步：访问A。

第**2**步：访问(A的邻接点)C。

在第1步访问A之后，接下来应该访问的是A的邻接点，即"C,D,F"中的一个。但在本文的实现中，顶点ABCDEFG是按照顺序存储，C在"D和F"的前面，因此，先访问C。

第**3**步：访问(C的邻接点)B。

在第2步访问C之后，接下来应该访问C的邻接点，即"B和D"中一个(A已经被访问过，就不算在内)。而由于B在D之前，先访问B。

第**4**步：访问(C的邻接点)D。

在第3步访问了C的邻接点B之后，B没有未被访问的邻接点；因此，返回到访问C的另一个邻接点D。

第**5**步：访问(A的邻接点)F。

前面已经访问了A，并且访问完了"A的邻接点B的所有邻接点(包括递归的邻接点在内)"；因此，此时返回到访问A的另一个邻接点F。

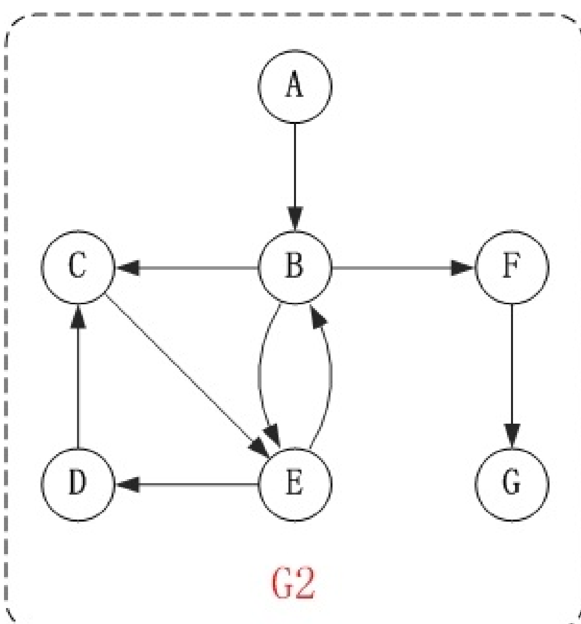
第**6**步：访问(F的邻接点)G。

第**7**步：访问(G的邻接点)E。

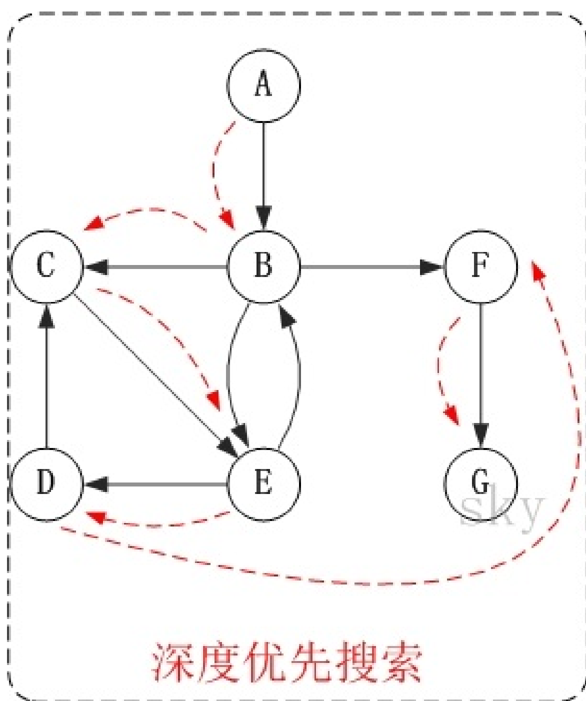
因此访问顺序是：**A -> C -> B -> D -> F -> G -> E**

2.2 有向图的深度优先搜索

下面以"有向图"为例，来对深度优先搜索进行演示。



对上面的图G2进行深度优先遍历，从顶点A开始。



第**1**步：访问A。

第**2**步：访问B。

在访问了A之后，接下来应该访问的是A的出边的另一个顶点，即顶点B。

第**3**步：访问C。

在访问了B之后，接下来应该访问的是B的出边的另一个顶点，即顶点C,E,F。在本文实现的图中，顶点ABCDEFG按照顺序存储，因此先访问C。

第**4**步：访问E。

接下来访问C的出边的另一个顶点，即顶点E。

第**5**步：访问D。

接下来访问E的出边的另一个顶点，即顶点B,D。顶点B已经被访问过，因此访问顶点D。

第**6**步：访问F。

接下应该回溯"访问A的出边的另一个顶点F"。

第**7**步：访问G。

因此访问顺序是：**A -> B -> C -> E -> D -> F -> G**

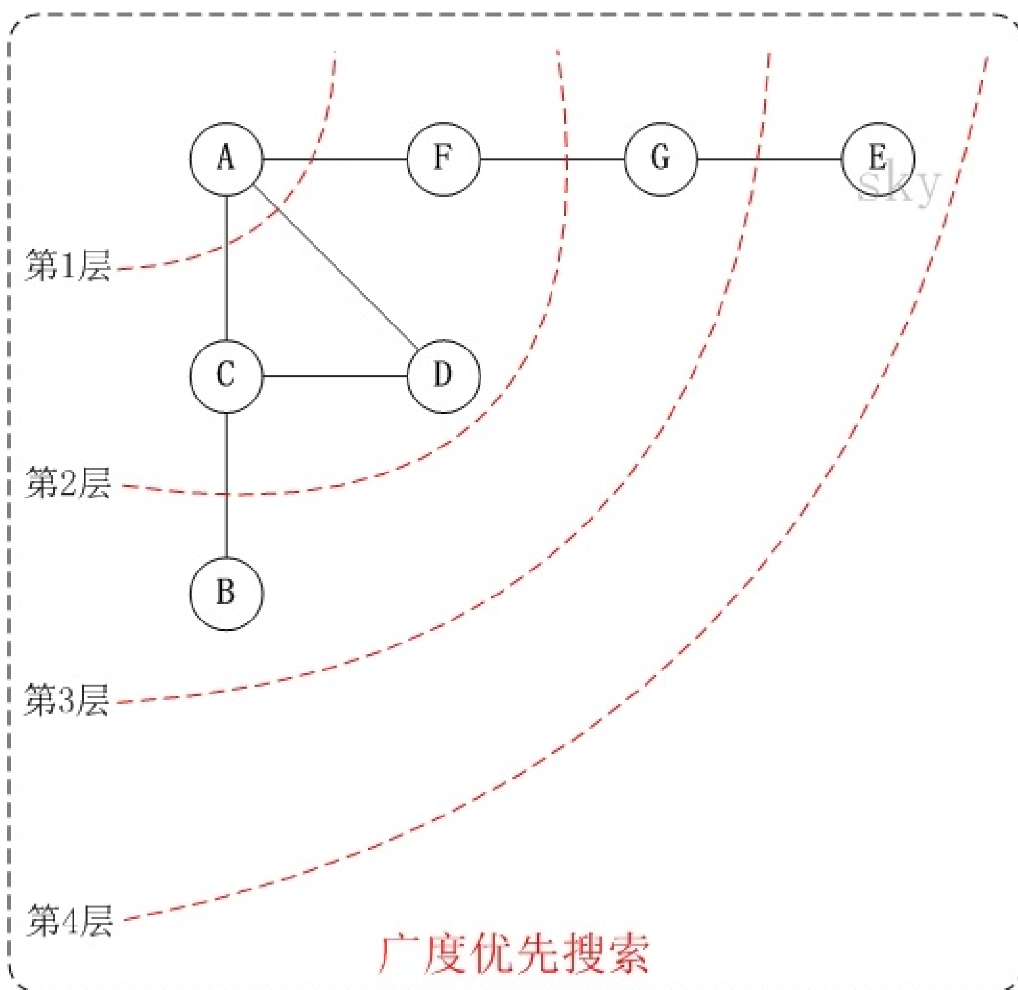
3. 广度优先搜索介绍

广度优先搜索算法(Breadth First Search)，又称为"宽度优先搜索"或"横向优先搜索"，简称BFS。

它的思想是：从图中某顶点v出发，在访问了v之后依次访问v的各个未曾访问过的邻接点，然后分别从这些邻接点出发依次访问它们的邻接点，并使得“先被访问的顶点的邻接点先于后被访问的顶点的邻接点被访问，直至图中所有已被访问的顶点的邻接点都被访问到。如果此时图中尚有顶点未被访问，则需要另选一个未曾被访问过的顶点作为新的起始点，重复上述过程，直至图中所有顶点都被访问到为止。

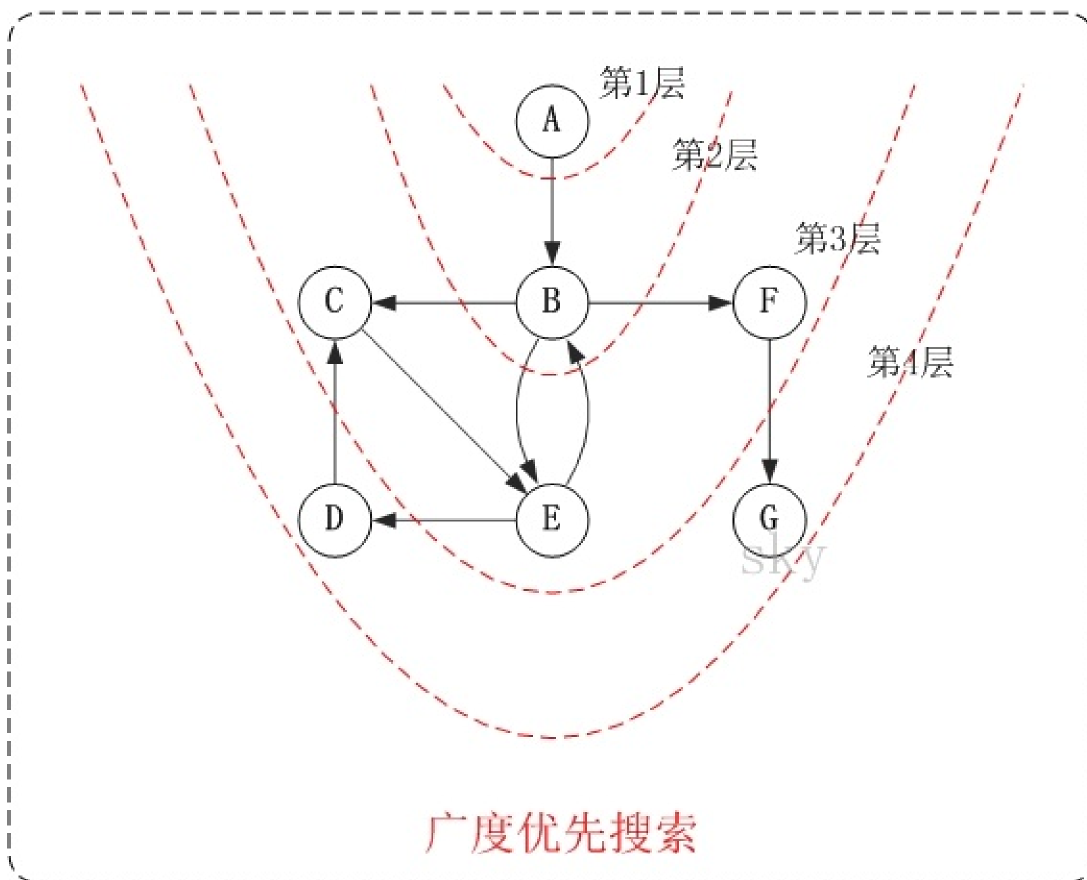
换句话说，广度优先搜索遍历图的过程是以v为起点，由近至远，依次访问和v有路径相通且路径长度为1,2...的顶点。

4. 广度优先搜索图解



4.1 无向图的广度优先搜索

下面以"无向图"为例，来对广度优先搜索进行演示。还是以上面的图G1为例进行说明。



第**1**步：访问A。

第**2**步：依次访问C,D,F。

在访问了A之后，接下来访问A的邻接点。前面已经说过，在本文实现中，顶点ABCDEFG按照顺序存储的，C在"D和F"的前面，因此，先访问C。再访问完C之后，再依次访问D,F。

第**3**步：依次访问B,G。

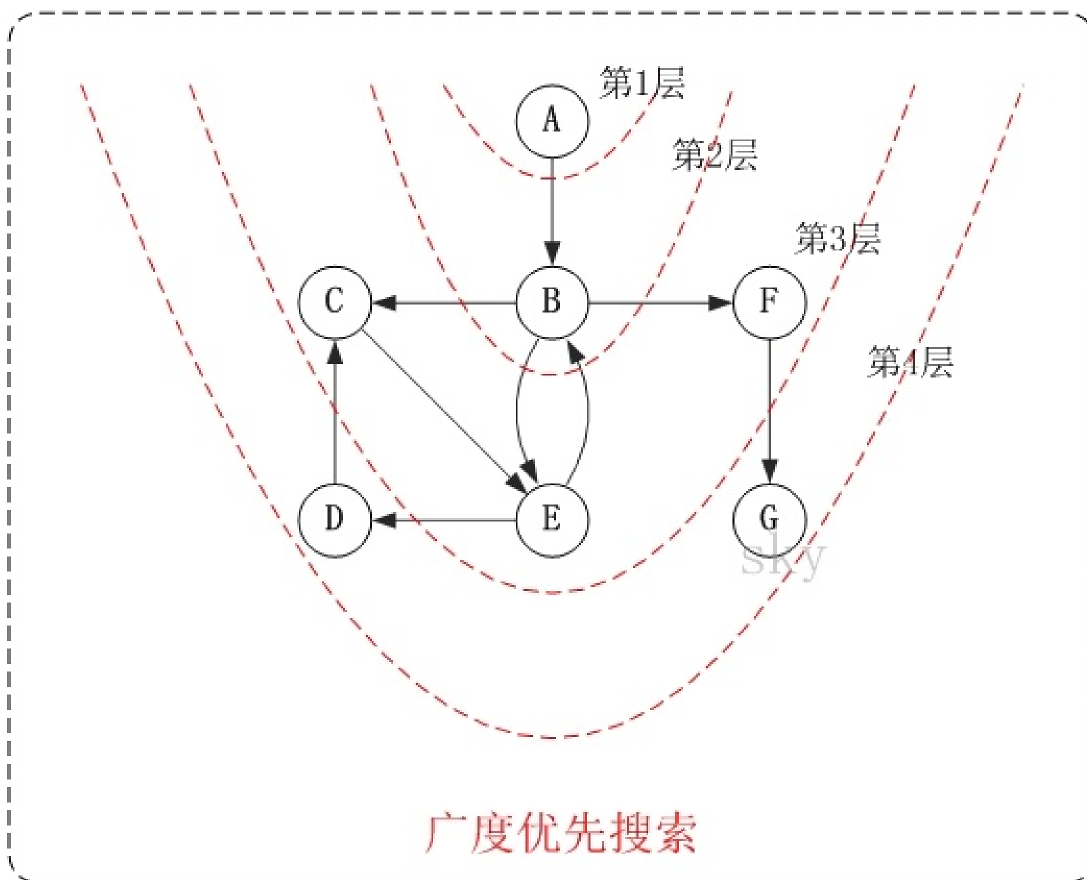
在第2步访问完C,D,F之后，再依次访问它们的邻接点。首先访问C的邻接点B，再访问F的邻接点G。

第**4**步：访问E。 在第3步访问完B,G之后，再依次访问它们的邻接点。只有G有邻接点E，因此访问G的邻接点E。

因此访问顺序是：**A -> C -> D -> F -> B -> G -> E**

4.2 有向图的广度优先搜索

下面以"有向图"为例，来对广度优先搜索进行演示。还是以上面的图G2为例进行说明。



第**1**步：访问A。

第**2**步：访问B。

第**3**步：依次访问C,E,F。

在访问了B之后，接下来访问B的出边的另一个顶点，即C,E,F。前面已经说过，在本文实现中，顶点ABCDEFG按照顺序存储的，因此会先访问C，再依次访问E,F。

第**4**步：依次访问D,G。 在访问完C,E,F之后，再依次访问它们的出边的另一个顶点。还是按照C,E,F的顺序访问，C的已经全部访问过了，那么就只剩下E,F；先访问E的邻接点D，再访问F的邻接点G。

因此访问顺序是：**A -> B -> C -> E -> F -> D -> G**

一、拓扑排序介绍

拓扑排序(Topological Order)是指，将一个有向无环图(Directed Acyclic Graph简称DAG)进行排序进而得到一个有序的线性序列。

这样说，可能理解起来比较抽象。下面通过简单的例子进行说明！

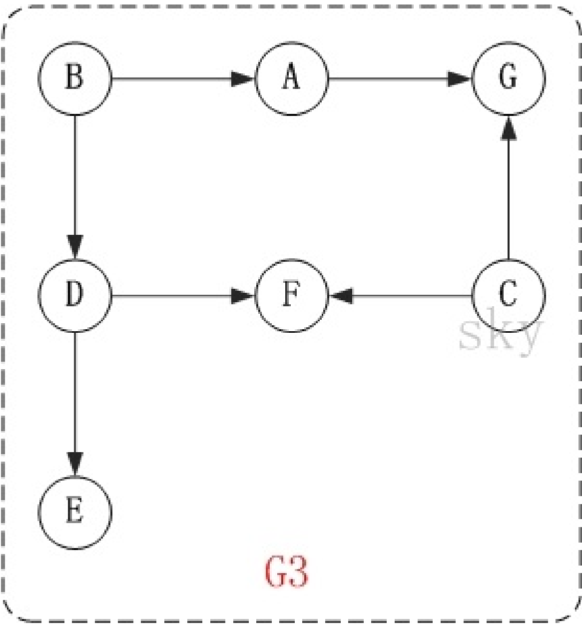
例如，一个项目包括A、B、C、D四个子部分来完成，并且A依赖于B和D，C依赖于D。现在要制定一个计划，写出A、B、C、D的执行顺序。这时，就可以利用到拓扑排序，它就是用来确定事物发生的顺序的。

在拓扑排序中，如果存在一条从顶点A到顶点B的路径，那么在排序结果中B出现在A的后面。

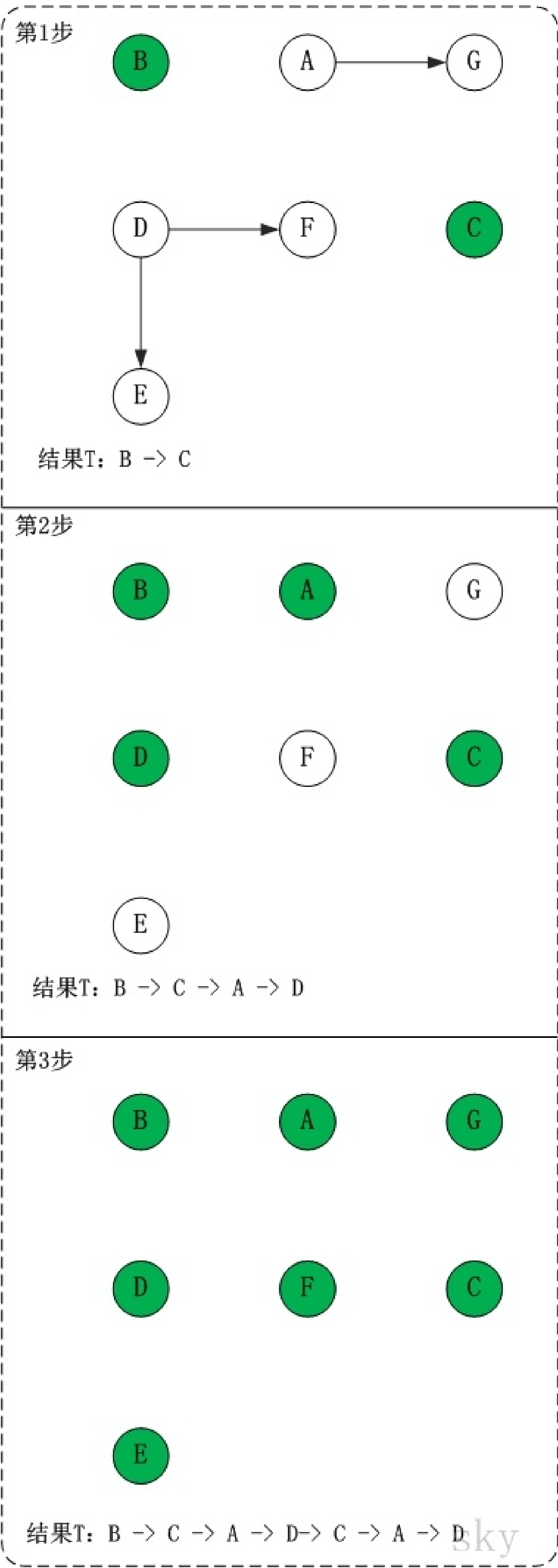
二、拓扑排序的算法图解

拓扑排序算法的基本步骤：

> **1.** 构造一个队列Q(queue) 和 拓扑排序的结果队列T(topological)；
>
> **2.** 把所有没有依赖顶点的节点放入Q；
>
> **3.** 当Q还有顶点的时候，执行下面步骤：
>
> **3.1** 从Q中取出一个顶点n(将n从Q中删掉)，并放入T(将n加入到结果集中)；
>
> **3.2** 对n每一个邻接点m(n是起点，m是终点)；
>
> **3.2.1** 去掉边;
>
> **3.2.2** 如果m没有依赖顶点，则把m放入Q;
>
> 注：顶点A没有依赖顶点，是指不存在以A为终点的边。



以上图为例，来对拓扑排序进行演示。


第1步
B
A
G
D
F
C
E
结果T：B -> C
第2步
B
A
G
D
F
C
E
结果T：B -> C -> A -> D
第3步
B
A
G
D
F
C
E
结果T：B -> C -> A -> D-> C -> A -> D

第**1**步：将B和C加入到排序结果中。

顶点B和顶点C都是没有依赖顶点，因此将C和C加入到结果集T中。假设ABCDEFG按顺序存储，因此先访问B，再访问C。访问B之后，去掉边和，并将A和D加入到队列Q中。同样的，去掉边和，并将F和G加入到Q中。

(01) 将B加入到排序结果中，然后去掉边和；此时，由于A和D没有依赖顶点，因此并将A和D加入到队列Q中。

(02) 将C加入到排序结果中，然后去掉边和；此时，由于F有依赖顶点D，G有依赖顶点A，因此不对F和G进行处理。

第**2**步：将A,D依次加入到排序结果中。

第1步访问之后，A,D都是没有依赖顶点的，根据存储顺序，先访问A，然后访问D。访问之后，删除顶点A和顶点D的出边。

第**3**步：将E,F,G依次加入到排序结果中。

因此访问顺序是：**B -> C -> A -> D -> E -> F -> G**

三、拓扑排序的代码说明

拓扑排序是对有向无向图的排序。下面以邻接表实现的有向图来对拓扑排序进行说明。

1. 基本定义

```
public class ListDG {
    // 邻接表中表对应的链表的顶点
    private class ENode {
        int ivex;        // 该边所指向的顶点的位置
        ENode nextEdge; // 指向下一条弧的指针
    }

    // 邻接表中表的顶点
    private class VNode {
        char data;          // 顶点信息
        ENode firstEdge;    // 指向第一条依附该顶点的弧
    };

    private VNode[] mVexs;  // 顶点数组

    ...
}
```

(01) ListDG是邻接表对应的结构体。 mVexs则是保存顶点信息的一维数组。

(02) VNode是邻接表顶点对应的结构体。 data是顶点所包含的数据，而firstEdge是该顶点所包含链表的表头指针。

(03) ENode是邻接表顶点所包含的链表的节点对应的结构体。 ivex是该节点所对应的顶点在vexs中的索引，而nextEdge是指向下一个节点的。

2. 拓扑排序

```
/*
 * 拓扑排序
 *
 * 返回值：
 *     -1 -- 失败(由于内存不足等原因导致)
 *      0 -- 成功排序，并输入结果
 *      1 -- 失败(该有向图是有环的)
 */
public int topologicalSort() {
    int index = 0;
    int num = mVexs.size();
```

```
    int[] ins;                 // 入度数组
    char[] tops;               // 拓扑排序结果数组，记录每个节点的排序后
的序号。
    Queue<Integer> queue;      // 辅组队列

    ins   = new int[num];
    tops  = new char[num];
    queue = new LinkedList<Integer>();

    // 统计每个顶点的入度数
    for(int i = 0; i < num; i++) {

        ENode node = mVexs.get(i).firstEdge;
        while (node != null) {
            ins[node.ivex]++;
            node = node.nextEdge;
        }
    }

    // 将所有入度为0的顶点入队列
    for(int i = 0; i < num; i ++)
        if(ins[i] == 0)
            queue.offer(i);                // 入队列

    while (!queue.isEmpty()) {             // 队列非空
        int j = queue.poll().intValue();   // 出队列。j是顶点的序号

        tops[index++] = mVexs.get(j).data;  // 将该顶点添加到tops中
，tops是排序结果
        ENode node = mVexs.get(j).firstEdge;// 获取以该顶点为起点的
出边队列

        // 将与"node"关联的节点的入度减1；
        // 若减1之后，该节点的入度为0；则将该节点添加到队列中。
        while(node != null) {
            // 将节点(序号为node.ivex)的入度减1。
            ins[node.ivex]--;
            // 若节点的入度为0，则将其"入队列"
            if( ins[node.ivex] == 0)
                queue.offer(node.ivex);    // 入队列
```

```
                node = node.nextEdge;
            }
        }

        if(index != num) {
            System.out.printf("Graph has a cycle\n");
            return 1;
        }

        // 打印拓扑排序结果
        System.out.printf("== TopSort: ");
        for(int i = 0; i < num; i ++)
            System.out.printf("%c ", tops[i]);
        System.out.printf("\n");

        return 0;
    }
```

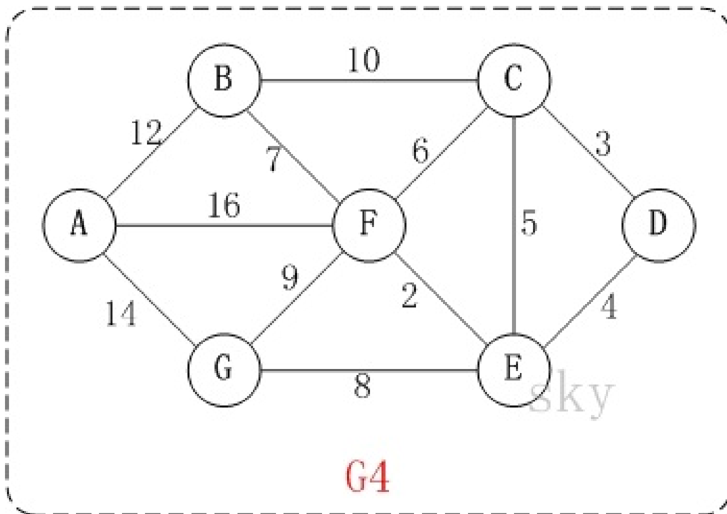
说明：

(01) queue的作用就是用来存储没有依赖顶点的顶点。它与前面所说的Q相对应。

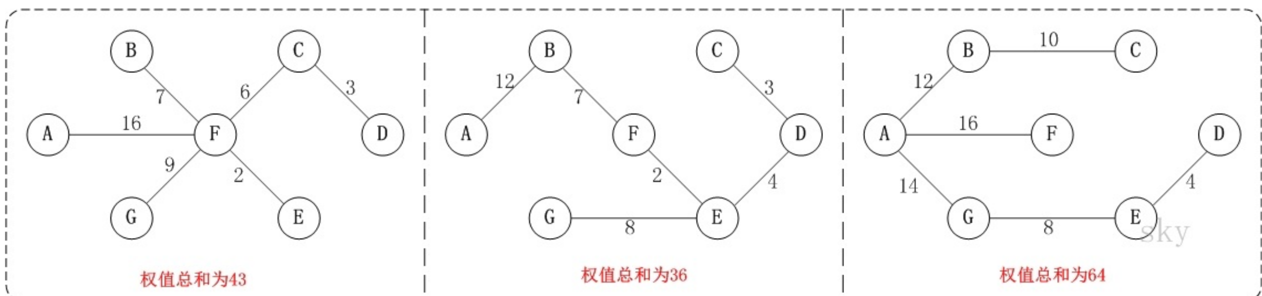
(02) tops的作用就是用来存储排序结果。它与前面所说的T相对应。

一、最小生成树

在含有n个顶点的连通图中选择n-1条边，构成一棵极小连通子图，并使该连通子图中n-1条边上权值之和达到最小，则称其为连通网的最小生成树。



例如，对于如上图G4所示的连通网可以有多棵权值总和不相同的生成树。



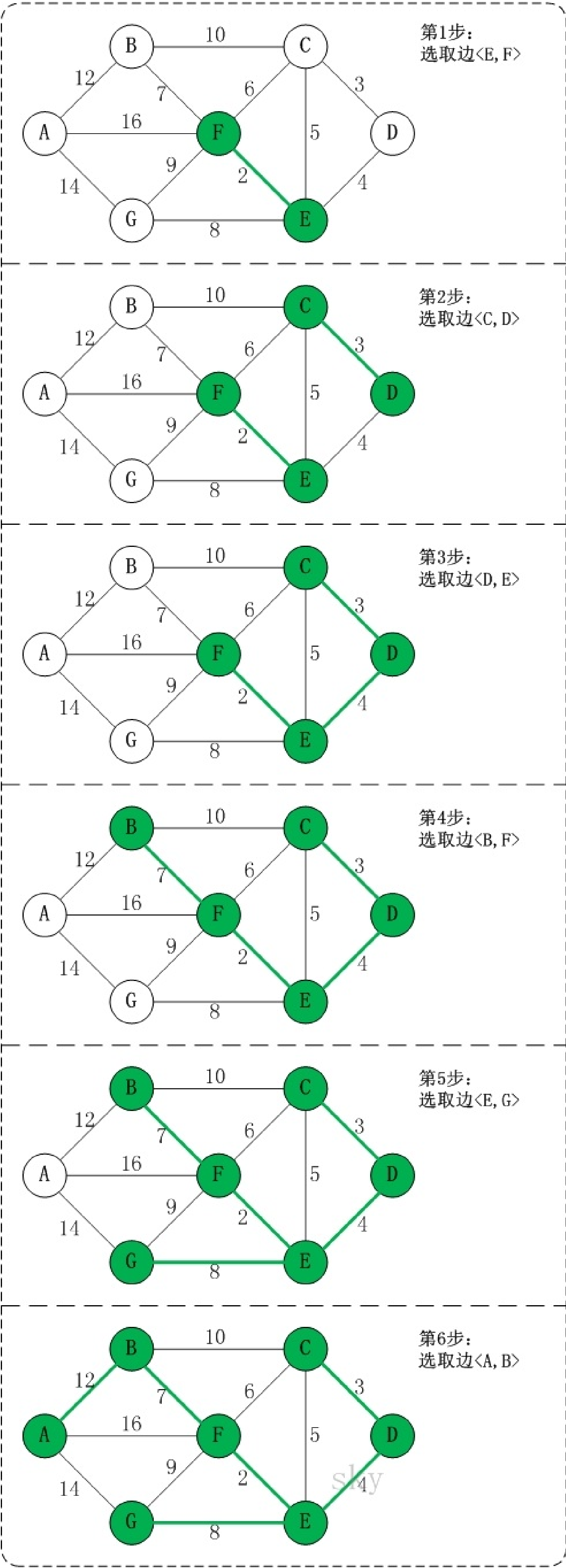
二、克鲁斯卡尔算法介绍

克鲁斯卡尔(Kruskal)算法，是用来求加权连通图的最小生成树的算法。

基本思想：按照权值从小到大的顺序选择n-1条边，并保证这n-1条边不构成回路。

具体做法：首先构造一个只含n个顶点的森林，然后依权值从小到大从连通网中选择边加入到森林中，并使森林中不产生回路，直至森林变成一棵树为止。

三、克鲁斯卡尔算法图解


第1步：
选取边<E, F>
第2步：
选取边<C, D>
第3步：
选取边<D, E>
第4步：
选取边<B, F>
第5步：
选取边<E, G>
第6步：
选取边<A, B>

以上图G4为例，来对克鲁斯卡尔进行演示(假设，用数组R保存最小生成树结果)。

第**1**步：将边加入R中。

 边的权值最小，因此将它加入到最小生成树结果R中。

第**2**步：将边加入R中。

 上一步操作之后，边的权值最小，因此将它加入到最小生成树结果R中。

第**3**步：将边加入R中。

 上一步操作之后，边的权值最小，因此将它加入到最小生成树结果R中。

第**4**步：将边加入R中。

 上一步操作之后，边的权值最小，但会和已有的边构成回路；因此，跳过边。同理，跳过边。将边加入到最小生成树结果R中。

第**5**步：将边加入R中。

 上一步操作之后，边的权值最小，因此将它加入到最小生成树结果R中。

第**6**步：将边加入R中。

 上一步操作之后，边的权值最小，但会和已有的边构成回路；因此，跳过边。同理，跳过边。将边加入到最小生成树结果R中。

此时，最小生成树构造完成！它包括的边依次是： 。

四、克鲁斯卡尔算法分析

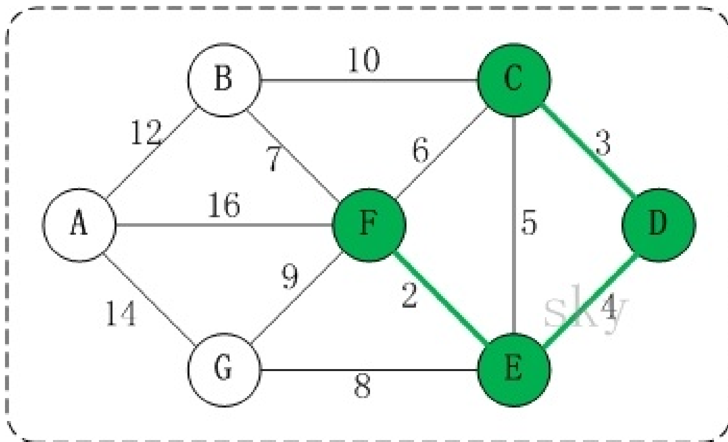
根据前面介绍的克鲁斯卡尔算法的基本思想和做法，我们能够了解到，克鲁斯卡尔算法重点需要解决的以下两个问题：

问题一 对图的所有边按照权值大小进行排序。

问题二 将边添加到最小生成树中时，怎么样判断是否形成了回路。

问题一很好解决，采用排序算法进行排序即可。

问题二，处理方式是：记录顶点在"最小生成树"中的终点，顶点的终点是"在最小生成树中与它连通的最大顶点"(关于这一点，后面会通过图片给出说明)。然后每次需要将一条边添加到最小生存树时，判断该边的两个顶点的终点是否重合，重合的话则会构成回路。 以下图来进行说明：



在将 加入到最小生成树R中之后，这几条边的顶点就都有了终点：

> **(01)** C的终点是F。
>
> **(02)** D的终点是F。
>
> **(03)** E的终点是F。
>
> **(04)** F的终点是F。

关于终点，就是将所有顶点按照从小到大的顺序排列好之后；某个顶点的终点就是"与它连通的最大顶点"。 因此，接下来，虽然是权值最小的边。但是C和E的重点都是F，即它们的终点相同，因此，将加入最小生成树的话，会形成回路。这就是判断回路的方式。

五、克鲁斯卡尔算法的代码说明

有了前面的算法分析之后，下面我们来查看具体代码。这里选取"邻接矩阵"进行说明，对于"邻接表"实现的图在后面的源码中会给出相应的源码。

1. 基本定义

```
// 边的结构体
private static class EData {
    char start; // 边的起点
    char end;   // 边的终点
    int weight; // 边的权重

    public EData(char start, char end, int weight) {
        this.start = start;
        this.end = end;
        this.weight = weight;
    }
};
```

EData是邻接矩阵边对应的结构体。

```
public class MatrixUDG {

    private int mEdgNum;        // 边的数量
    private char[] mVexs;       // 顶点集合
    private int[][] mMatrix;    // 邻接矩阵
    private static final int INF = Integer.MAX_VALUE;   // 最大值

    ...
}
```

MatrixUDG是邻接矩阵对应的结构体。mVexs用于保存顶点，mEdgNum用于保存边数，mMatrix则是用于保存矩阵信息的二维数组。例如，mMatrix[i][j]=1，则表示"顶点i(即mVexs[i])"和"顶点j(即mVexs[j])"是邻接点；mMatrix[i][j]=0，则表示它们不是邻接点。

2. 克鲁斯卡尔算法

```
/*
 * 克鲁斯卡尔（Kruskal)最小生成树
 */
public void kruskal() {
    int index = 0;                      // rets数组的索引
    int[] vends = new int[mEdgNum];     // 用于保存"已有最小生成树"
```

```
中每个顶点在该最小树中的终点。
    EData[] rets = new EData[mEdgNum];  // 结果数组，保存kruskal最
小生成树的边
    EData[] edges;                      // 图对应的所有边

    // 获取"图中所有的边"
    edges = getEdges();
    // 将边按照"权"的大小进行排序(从小到大)
    sortEdges(edges, mEdgNum);

    for (int i=0; i<mEdgNum; i++) {
        int p1 = getPosition(edges[i].start);      // 获取第i条边
的"起点"的序号
        int p2 = getPosition(edges[i].end);        // 获取第i条边
的"终点"的序号

        int m = getEnd(vends, p1);                 // 获取p1在"已
有的最小生成树"中的终点
        int n = getEnd(vends, p2);                 // 获取p2在"已
有的最小生成树"中的终点
        // 如果m!=n，意味着"边i"与"已经添加到最小生成树中的顶点"没有形成
环路
        if (m != n) {
            vends[m] = n;                       // 设置m在"已有的
最小生成树"中的终点为n
            rets[index++] = edges[i];           // 保存结果
        }
    }

    // 统计并打印"kruskal最小生成树"的信息
    int length = 0;
    for (int i = 0; i < index; i++)
        length += rets[i].weight;
    System.out.printf("Kruskal=%d: ", length);
    for (int i = 0; i < index; i++)
        System.out.printf("(%c,%c) ", rets[i].start, rets[i].end
);
    System.out.printf("\n");
}
```

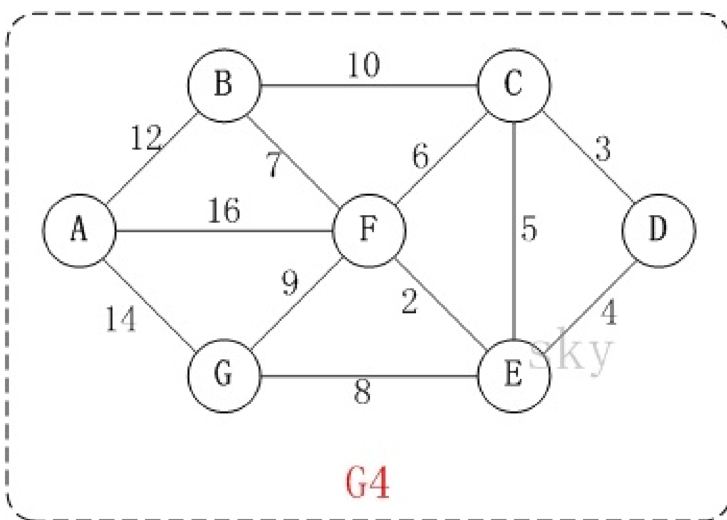
一、普里姆算法介绍

普里姆(Prim)算法，是用来求加权连通图的最小生成树的算法。

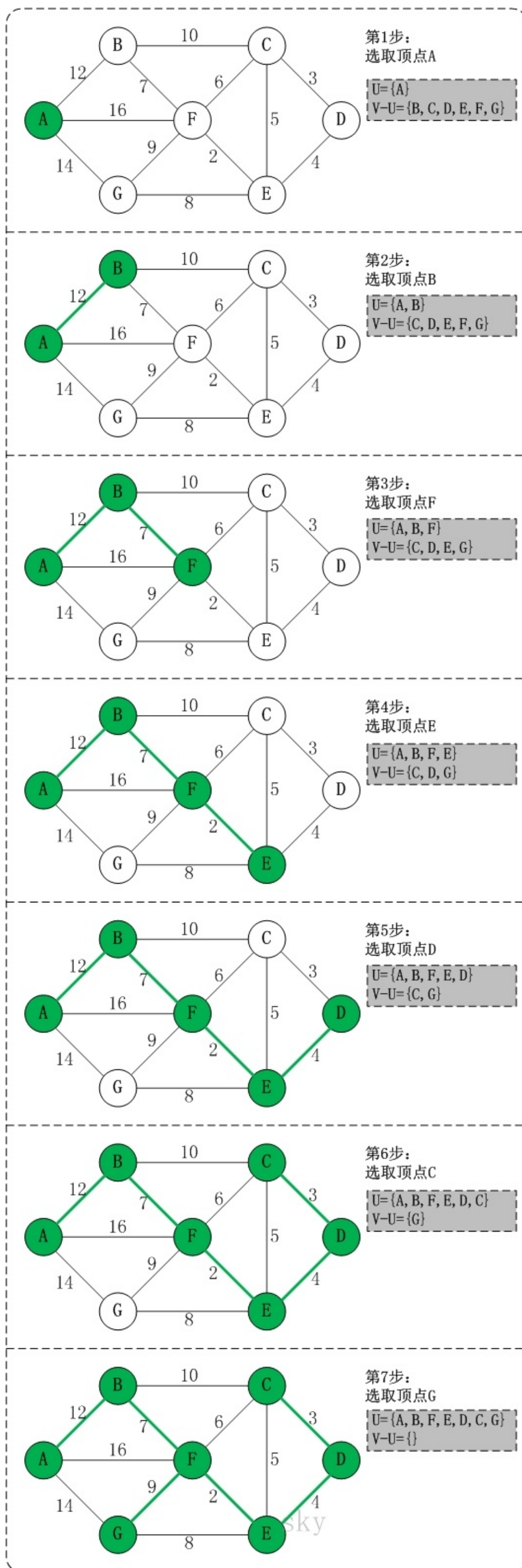
基本思想

对于图G而言，V是所有顶点的集合；现在，设置两个新的集合U和T，其中U用于存放G的最小生成树中的顶点，T存放G的最小生成树中的边。 从所有uЄU，vЄ(V-U) (V-U表示出去U的所有顶点)的边中选取权值最小的边(u, v)，将顶点v加入集合U中，将边(u, v)加入集合T中，如此不断重复，直到U=V为止，最小生成树构造完毕，这时集合T中包含了最小生成树中的所有边。

二、普里姆算法图解



以上图G4为例，来对普里姆进行演示(从第一个顶点A开始通过普里姆算法生成最小生成树)。


第1步:
选取顶点A
U={A}
V-U={B, C, D, E, F, G}
第2步:
选取顶点B
U={A, B}
V-U={C, D, E, F, G}
第3步:
选取顶点F
U={A, B, F}
V-U={C, D, E, G}
第4步:
选取顶点E
U={A, B, F, E}
V-U={C, D, G}
第5步:
选取顶点D
U={A, B, F, E, D}
V-U={C, G}
第6步:
选取顶点C
U={A, B, F, E, D, C}
V-U={G}
第7步:
选取顶点G
U={A, B, F, E, D, C, G}
V-U={}

初始状态：V是所有顶点的集合，即V={A,B,C,D,E,F,G}；U和T都是空！

第**1**步：将顶点A加入到U中。

此时，U={A}。

第**2**步：将顶点B加入到U中。

上一步操作之后，U={A}, V-U={B,C,D,E,F,G}；因此，边(A,B)的权值最小。将顶点B添加到U中；此时，U={A,B}。

第**3**步：将顶点F加入到U中。

上一步操作之后，U={A,B}, V-U={C,D,E,F,G}；因此，边(B,F)的权值最小。将顶点F添加到U中；此时，U={A,B,F}。

第**4**步：将顶点E加入到U中。

上一步操作之后，U={A,B,F}, V-U={C,D,E,G}；因此，边(F,E)的权值最小。将顶点E添加到U中；此时，U={A,B,F,E}。

第**5**步：将顶点D加入到U中。

上一步操作之后，U={A,B,F,E}, V-U={C,D,G}；因此，边(E,D)的权值最小。将顶点D添加到U中；此时，U={A,B,F,E,D}。

第**6**步：将顶点C加入到U中。

上一步操作之后，U={A,B,F,E,D}, V-U={C,G}；因此，边(D,C)的权值最小。将顶点C添加到U中；此时，U={A,B,F,E,D,C}。

第**7**步：将顶点G加入到U中。

上一步操作之后，U={A,B,F,E,D,C}, V-U={G}；因此，边(F,G)的权值最小。将顶点G添加到U中；此时，U=V。

此时，最小生成树构造完成！它包括的顶点依次是：**A B F E D C G**。

三、普里姆算法的代码说明

以"邻接矩阵"为例对普里姆算法进行说明。

1. 基本定义

```
public class MatrixUDG {

    private char[] mVexs;       // 顶点集合
    private int[][] mMatrix;    // 邻接矩阵
    private static final int INF = Integer.MAX_VALUE;   // 最大值

    ...
}
```

MatrixUDG是邻接矩阵对应的结构体。mVexs用于保存顶点，mEdgNum用于保存边数，mMatrix则是用于保存矩阵信息的二维数组。例如，mMatrix[i][j]=1，则表示"顶点i(即mVexs[i])"和"顶点j(即mVexs[j])"是邻接点；mMatrix[i][j]=0，则表示它们不是邻接点。

2. 普里姆算法

```
/*
 * prim最小生成树
 *
 * 参数说明：
 *   start -- 从图中的第start个元素开始，生成最小树
 */
public void prim(int start) {
    int num = mVexs.length;         // 顶点个数
    int index=0;                    // prim最小树的索引，即prims数
组的索引
    char[] prims  = new char[num];  // prim最小树的结果数组
    int[] weights = new int[num];   // 顶点间边的权值

    // prim最小生成树中第一个数是"图中第start个顶点"，因为是从start开始
的。
    prims[index++] = mVexs[start];

    // 初始化"顶点的权值数组"，
    // 将每个顶点的权值初始化为"第start个顶点"到"该顶点"的权值。
    for (int i = 0; i < num; i++ )
        weights[i] = mMatrix[start][i];
    // 将第start个顶点的权值初始化为0。
    // 可以理解为"第start个顶点到它自身的距离为0"。
```

```
    weights[start] = 0;

    for (int i = 0; i < num; i++) {
        // 由于从start开始的，因此不需要再对第start个顶点进行处理。
        if(start == i)
            continue;

        int j = 0;
        int k = 0;
        int min = INF;
        // 在未被加入到最小生成树的顶点中，找出权值最小的顶点。
        while (j < num) {
            // 若weights[j]=0，意味着"第j个节点已经被排序过"(或者说已经
加入了最小生成树中)。
            if (weights[j] != 0 && weights[j] < min) {
                min = weights[j];
                k = j;
            }
            j++;
        }

        // 经过上面的处理后，在未被加入到最小生成树的顶点中，权值最小的顶点
是第k个顶点。
        // 将第k个顶点加入到最小生成树的结果数组中
        prims[index++] = mVexs[k];
        // 将"第k个顶点的权值"标记为0，意味着第k个顶点已经排序过了(或者说
已经加入了最小树结果中)。
        weights[k] = 0;
        // 当第k个顶点被加入到最小生成树的结果数组中之后，更新其它顶点的权
值。
        for (j = 0 ; j < num; j++) {
            // 当第j个节点没有被处理，并且需要更新时才被更新。
            if (weights[j] != 0 && mMatrix[k][j] < weights[j])
                weights[j] = mMatrix[k][j];
        }
    }

    // 计算最小生成树的权值
    int sum = 0;
    for (int i = 1; i < index; i++) {
```

```
            int min = INF;
            // 获取prims[i]在mMatrix中的位置
            int n = getPosition(prims[i]);
            // 在vexs[0...i]中，找出到j的权值最小的顶点。
            for (int j = 0; j < i; j++) {
                int m = getPosition(prims[j]);
                if (mMatrix[m][n]<min)
                    min = mMatrix[m][n];
            }
            sum += min;
        }
        // 打印最小生成树
        System.out.printf("PRIM(%c)=%d: ", mVexs[start], sum);
        for (int i = 0; i < index; i++)
            System.out.printf("%c ", prims[i]);
        System.out.printf("\n");
    }
```

一、迪杰斯特拉算法介绍

迪杰斯特拉(Dijkstra)算法是典型最短路径算法，用于计算一个节点到其他节点的最短路径。 它的主要特点是以起始点为中心向外层层扩展(广度优先搜索思想)，直到扩展到终点为止。

基本思想

通过Dijkstra计算图G中的最短路径时，需要指定起点s(即从顶点s开始计算)。

此外，引进两个集合S和U。S的作用是记录已求出最短路径的顶点(以及相应的最短路径长度)，而U则是记录还未求出最短路径的顶点(以及该顶点到起点s的距离)。

初始时，S中只有起点s；U中是除s之外的顶点，并且U中顶点的路径是"起点s到该顶点的路径"。然后，从U中找出路径最短的顶点，并将其加入到S中；接着，更新U中的顶点和顶点对应的路径。 然后，再从U中找出路径最短的顶点，并将其加入到S中；接着，更新U中的顶点和顶点对应的路径。 ... 重复该操作，直到遍历完所有顶点。

操作步骤

(1) 初始时，S只包含起点s；U包含除s外的其他顶点，且U中顶点的距离为"起点s到该顶点的距离"[例如，U中顶点v的距离为(s,v)的长度，然后s和v不相邻，则v的距离为∞]。

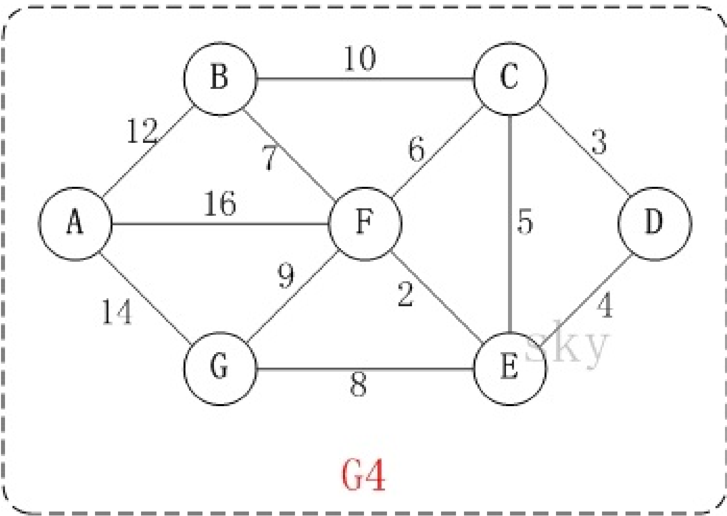
(2) 从U中选出"距离最短的顶点k"，并将顶点k加入到S中；同时，从U中移除顶点k。

(3) 更新U中各个顶点到起点s的距离。之所以更新U中顶点的距离，是由于上一步中确定了k是求出最短路径的顶点，从而可以利用k来更新其它顶点的距离；例如，(s,v)的距离可能大于(s,k)+(k,v)的距离。

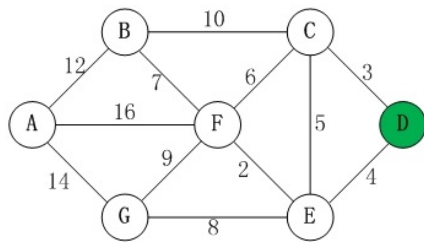
(4) 重复步骤(2)和(3)，直到遍历完所有顶点。

单纯的看上面的理论可能比较难以理解，下面通过实例来对该算法进行说明。

二、迪杰斯特拉算法图解



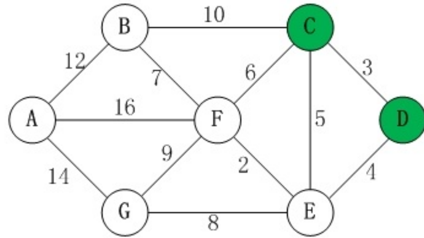
以上图G4为例，来对迪杰斯特拉进行算法演示(以第4个顶点D为起点)。



第1步：
选取顶点D

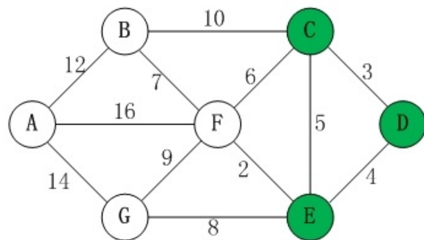
S={D(0)}
U={A(∞),B(∞),C(3),E(4),F(∞),G(∞)}

注：
(01) S是已计算出最短路径的定点的集合
(02) U是未计算出最短路径的定点的集合
(03) C(3)表示顶点C到起点D的最短距离是3



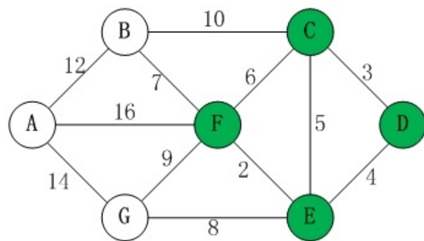
第2步：
选取顶点C

S={D(0),C(3)}
U={A(∞),B(23),E(4),F(9),G(∞)}



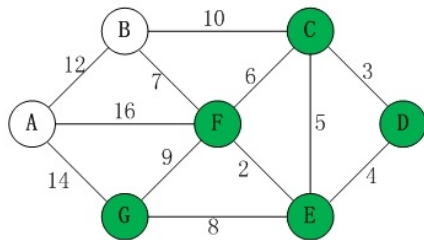
第3步：
选取顶点E

S={D(0),C(3),E(4)}
U={A(∞),B(23),F(6),G(12)}



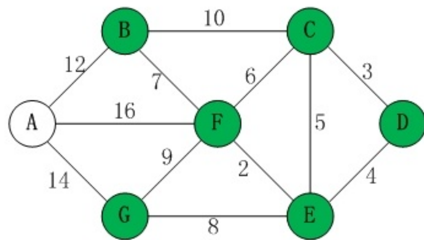
第4步：
选取顶点F

S={D(0),C(3),E(4),F(6)}
U={A(22),B(13),G(12)}



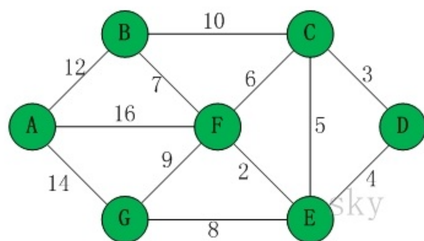
第5步：
选取顶点G

S={D(0),C(3),E(4),F(6),G(12)}
U={A(22),B(13)}



第6步：
选取顶点B

S={D(0),C(3),E(4),F(6),G(12),B(13)}
U={A(22)}



第7步：
选取顶点A

S={D(0),C(3),E(4),F(6),G(12),B(13),A(22)}

初始状态：S是已计算出最短路径的顶点集合，U是未计算除最短路径的顶点的集合！

第**1**步：将顶点D加入到S中。

此时，S={D(0)}, U={A(∞),B(∞),C(3),E(4),F(∞),G(∞)}。 注:C(3)表示C到起点D的距离是3。

第**2**步：将顶点C加入到S中。

上一步操作之后，U中顶点C到起点D的距离最短；因此，将C加入到S中，同时更新U中顶点的距离。以顶点F为例，之前F到D的距离为∞；但是将C加入到S之后，F到D的距离为9=(F,C)+(C,D)。

此时，S={D(0),C(3)}, U={A(∞),B(23),E(4),F(9),G(∞)}。

第**3**步：将顶点E加入到S中。

上一步操作之后，U中顶点E到起点D的距离最短；因此，将E加入到S中，同时更新U中顶点的距离。还是以顶点F为例，之前F到D的距离为9；但是将E加入到S之后，F到D的距离为6=(F,E)+(E,D)。

此时，S={D(0),C(3),E(4)}, U={A(∞),B(23),F(6),G(12)}。

第**4**步：将顶点F加入到S中。

此时，S={D(0),C(3),E(4),F(6)}, U={A(22),B(13),G(12)}。

第**5**步：将顶点G加入到S中。

此时，S={D(0),C(3),E(4),F(6),G(12)}, U={A(22),B(13)}。

第**6**步：将顶点B加入到S中。

此时，S={D(0),C(3),E(4),F(6),G(12),B(13)}, U={A(22)}。

第**7**步：将顶点A加入到S中。

此时，S={D(0),C(3),E(4),F(6),G(12),B(13),A(22)}。

此时，起点D到各个顶点的最短距离就计算出来了：**A(22) B(13) C(3) D(0) E(4) F(6) G(12)**。

三、迪杰斯特拉算法的代码说明

以"邻接矩阵"为例对迪杰斯特拉算法进行说明。

1. 基本定义

```
public class MatrixUDG {

    private int mEdgNum;        // 边的数量
    private char[] mVexs;       // 顶点集合
    private int[][] mMatrix;    // 邻接矩阵
    private static final int INF = Integer.MAX_VALUE;   // 最大值

    ...
}
```

MatrixUDG是邻接矩阵对应的结构体。mVexs用于保存顶点，mEdgNum用于保存边数，mMatrix则是用于保存矩阵信息的二维数组。例如，mMatrix[i][j]=1，则表示"顶点i(即mVexs[i])"和"顶点j(即mVexs[j])"是邻接点；mMatrix[i][j]=0，则表示它们不是邻接点。

2. 迪杰斯特拉算法

```
/*
 * Dijkstra最短路径。
 * 即，统计图中"顶点vs"到其它各个顶点的最短路径。
 *
 * 参数说明：
 *       vs -- 起始顶点(start vertex)。即计算"顶点vs"到其它顶点的最短
路径。
 *     prev -- 前驱顶点数组。即，prev[i]的值是"顶点vs"到"顶点i"的最短路
径所经历的全部顶点中，位于"顶点i"之前的那个顶点。
 *     dist -- 长度数组。即，dist[i]是"顶点vs"到"顶点i"的最短路径的长度
。
 */
public void dijkstra(int vs, int[] prev, int[] dist) {
    // flag[i]=true表示"顶点vs"到"顶点i"的最短路径已成功获取
    boolean[] flag = new boolean[mVexs.length];
```

```
    // 初始化
    for (int i = 0; i < mVexs.length; i++) {
        flag[i] = false;          // 顶点i的最短路径还没获取到。
        prev[i] = 0;              // 顶点i的前驱顶点为0。
        dist[i] = mMatrix[vs][i];  // 顶点i的最短路径为"顶点vs"到"顶
点i"的权。
    }

    // 对"顶点vs"自身进行初始化
    flag[vs] = true;
    dist[vs] = 0;

    // 遍历mVexs.length-1次；每次找出一个顶点的最短路径。
    int k=0;
    for (int i = 1; i < mVexs.length; i++) {
        // 寻找当前最小的路径；
        // 即，在未获取最短路径的顶点中，找到离vs最近的顶点(k)。
        int min = INF;
        for (int j = 0; j < mVexs.length; j++) {
            if (flag[j]==false && dist[j]<min) {
                min = dist[j];
                k = j;
            }
        }
        // 标记"顶点k"为已经获取到最短路径
        flag[k] = true;

        // 修正当前最短路径和前驱顶点
        // 即，当已经"顶点k的最短路径"之后，更新"未获取最短路径的顶点的最
短路径和前驱顶点"。
        for (int j = 0; j < mVexs.length; j++) {
            int tmp = (mMatrix[k][j]==INF ? INF : (min + mMatrix
[k][j]));
            if (flag[j]==false && (tmp<dist[j]) ) {
                dist[j] = tmp;
                prev[j] = k;
            }
        }
    }
```

```
    // 打印dijkstra最短路径的结果
    System.out.printf("dijkstra(%c): \n", mVexs[vs]);
    for (int i=0; i < mVexs.length; i++)
        System.out.printf("  shortest(%c, %c)=%d\n", mVexs[vs],
mVexs[i], dist[i]);
}
```

一、弗洛伊德算法介绍

和Dijkstra算法一样，弗洛伊德(Floyd)算法也是一种用于寻找给定的加权图中顶点间最短路径的算法。该算法名称以创始人之一、1978年图灵奖获得者、斯坦福大学计算机科学系教授罗伯特·弗洛伊德命名。

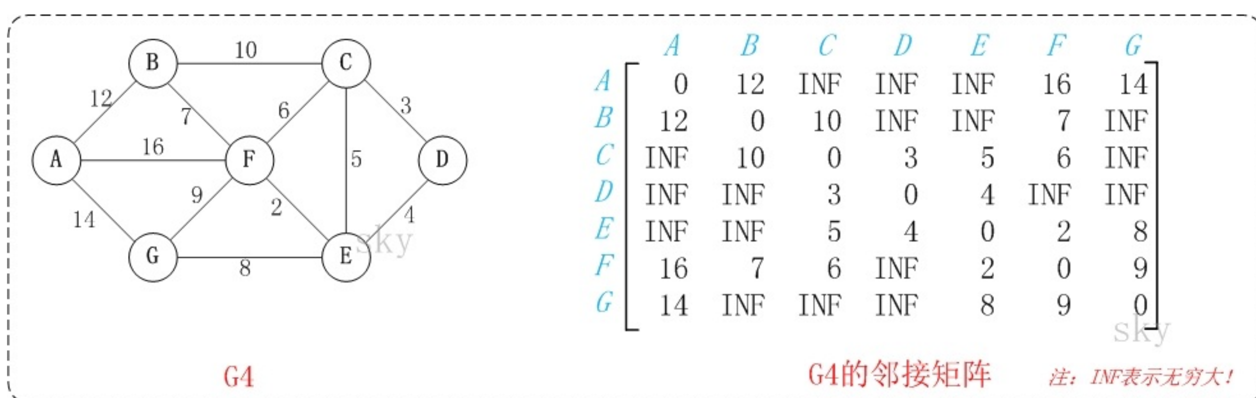
基本思想

通过Floyd计算图G=(V,E)中各个顶点的最短路径时，需要引入一个矩阵S，矩阵S中的元素 `a[i][j]` 表示顶点i(第i个顶点)到顶点j(第j个顶点)的距离。

假设图G中顶点个数为N，则需要对矩阵S进行N次更新。初始时，矩阵S中顶点 `a[i][j]` 的距离为顶点i到顶点j的权值；如果i和j不相邻，则 `a[i][j]=∞` 。 接下来开始，对矩阵S进行N次更新。第1次更新时，如果" `a[i][j]` 的距离" > " `a[i][0]+a[0][j]` "(`a[i][0]+a[0][j]` 表示"i与j之间经过第1个顶点的距离")，则更新 `a[i][j]` 为" `a[i][0]+a[0][j]` "。 同理，第k次更新时，如果" `a[i][j]` 的距离" > " `a[i][k]+a[k][j]` "，则更新 `a[i][j]` 为" `a[i][k]+a[k][j]` "。更新N次之后，操作完成！

单纯的看上面的理论可能比较难以理解，下面通过实例来对该算法进行说明。

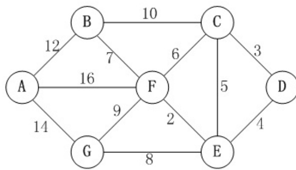
二、弗洛伊德算法图解



	A	B	C	D	E	F	G
A	0	12	INF	INF	INF	16	14
B	12	0	10	INF	INF	7	INF
C	INF	10	0	3	5	6	INF
D	INF	INF	3	0	4	INF	INF
E	INF	INF	5	4	0	2	8
F	16	7	6	INF	2	0	9
G	14	INF	INF	INF	8	9	0

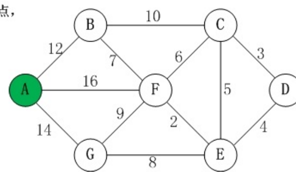
以上图G4为例，来对弗洛伊德进行算法演示。

第1步：
初始化矩阵S



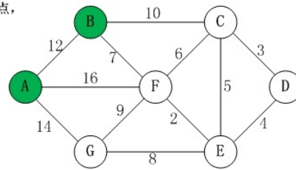
	A	B	C	D	E	F	G
A	0	12	INF	INF	INF	16	14
B	12	0	10	INF	INF	7	INF
C	INF	10	0	3	5	6	INF
D	INF	INF	3	0	4	INF	INF
E	INF	INF	5	4	0	2	8
F	16	7	6	INF	2	0	9
G	14	INF	INF	INF	8	9	0

第2步：
以顶点A为中介点，
更新矩阵S。



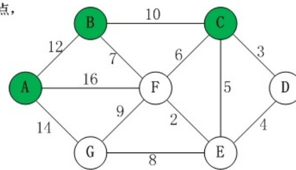
	A	B	C	D	E	F	G
A	0	12	INF	INF	INF	16	14
B	12	0	10	INF	INF	7	26
C	INF	10	0	3	5	6	INF
D	INF	INF	3	0	4	INF	INF
E	INF	INF	5	4	0	2	8
F	16	7	6	INF	2	0	9
G	14	26	INF	INF	8	9	0

第3步：
以顶点B为中介点，
更新矩阵S。



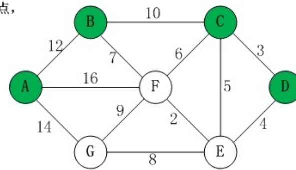
	A	B	C	D	E	F	G
A	0	12	22	INF	INF	16	14
B	12	0	10	INF	INF	7	26
C	22	10	0	3	5	6	36
D	INF	INF	3	0	4	INF	INF
E	INF	INF	5	4	0	2	8
F	16	7	6	INF	2	0	9
G	14	26	36	INF	8	9	0

第4步：
以顶点C为中介点，
更新矩阵S。



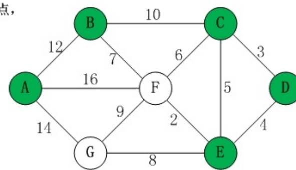
	A	B	C	D	E	F	G
A	0	12	22	25	27	16	14
B	12	0	10	13	15	7	26
C	22	10	0	3	5	6	36
D	25	13	3	0	4	9	39
E	27	15	5	4	0	2	8
F	16	7	6	9	2	0	9
G	14	26	36	39	8	9	0

第5步：
以顶点D为中介点，
更新矩阵S。



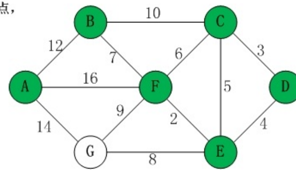
	A	B	C	D	E	F	G
A	0	12	22	25	27	16	14
B	12	0	10	13	15	7	26
C	22	10	0	3	5	6	36
D	25	13	3	0	4	9	39
E	27	15	5	4	0	2	8
F	16	7	6	9	2	0	9
G	14	26	36	39	8	9	0

第6步：
以顶点E为中介点，
更新矩阵S。



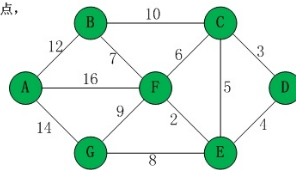
	A	B	C	D	E	F	G
A	0	12	22	25	27	16	14
B	12	0	10	13	15	7	23
C	22	10	0	3	5	6	13
D	25	13	3	0	4	6	12
E	27	15	5	4	0	2	8
F	16	7	6	6	2	0	9
G	14	23	13	12	8	9	0

第7步：
以顶点F为中介点，
更新矩阵S。



	A	B	C	D	E	F	G
A	0	12	22	22	18	16	14
B	12	0	10	13	9	7	16
C	22	10	0	3	5	6	13
D	22	13	3	0	4	6	12
E	18	9	5	4	0	2	8
F	16	7	6	6	2	0	9
G	14	16	13	12	8	9	0

第8步：
以顶点G为中介点，
更新矩阵S。



	A	B	C	D	E	F	G
A	0	12	22	22	18	16	14
B	12	0	10	13	9	7	16
C	22	10	0	3	5	6	13
D	22	13	3	0	4	6	12
E	18	9	5	4	0	2	8
F	16	7	6	6	2	0	9
G	14	16	13	12	8	9	0

初始状态：S是记录各个顶点间最短路径的矩阵。

第**1**步：初始化S。

矩阵S中顶点a[i][j]的距离为顶点i到顶点j的权值；如果i和j不相邻，则 `a[i][j]=∞` 。实际上，就是将图的原始矩阵复制到S中。

注: `a[i][j]` 表示矩阵S中顶点i(第i个顶点)到顶点j(第j个顶点)的距离。

第**2**步：以顶点A(第1个顶点)为中介点，若 `a[i][j] > a[i][0]+a[0][j]` ，则设置 `a[i][j]=a[i][0]+a[0][j]` 。 以顶点 `a[1][6]` ，上一步操作之后， `a[1][6]=∞` ；而将A作为中介点时，(B,A)=12，(A,G)=14，因此B和G之间的距离可以更新为16。

同理，依次将顶点B,C,D,E,F,G作为中介点，并更新 `a[i][j]` 的大小。

三、弗洛伊德算法的代码说明

以"邻接矩阵"为例对弗洛伊德算法进行说明，对于"邻接表"实现的图在后面会给出相应的源码。

1. 基本定义

```
public class MatrixUDG {

    private int mEdgNum;        // 边的数量
    private char[] mVexs;       // 顶点集合
    private int[][] mMatrix;    // 邻接矩阵
    private static final int INF = Integer.MAX_VALUE;   // 最大值

    ...
}
```

MatrixUDG是邻接矩阵对应的结构体。mVexs用于保存顶点，mEdgNum用于保存边数，mMatrix则是用于保存矩阵信息的二维数组。例如，mMatrix[i][j]=1，则表示"顶点i(即mVexs[i])"和"顶点j(即mVexs[j])"是邻接点；mMatrix[i][j]=0，则表示它们不是邻接点。

2. 弗洛伊德算法

```
/*
 * floyd最短路径。
 * 即，统计图中各个顶点间的最短路径。
 *
 * 参数说明：
 *     path -- 路径。path[i][j]=k表示，"顶点i"到"顶点j"的最短路径会经
过顶点k。
 *     dist -- 长度数组。即，dist[i][j]=sum表示，"顶点i"到"顶点j"的最
短路径的长度是sum。
 */
public void floyd(int[][] path, int[][] dist) {

    // 初始化
    for (int i = 0; i < mVexs.length; i++) {
        for (int j = 0; j < mVexs.length; j++) {
            dist[i][j] = mMatrix[i][j];    // "顶点i"到"顶点j"的路
径长度为"i到j的权值"。
            path[i][j] = j;                // "顶点i"到"顶点j"的最
短路径是经过顶点j。
        }
    }

    // 计算最短路径
    for (int k = 0; k < mVexs.length; k++) {
        for (int i = 0; i < mVexs.length; i++) {
            for (int j = 0; j < mVexs.length; j++) {

                // 如果经过下标为k顶点路径比原两点间路径更短，则更新dist
[i][j]和path[i][j]
                int tmp = (dist[i][k]==INF || dist[k][j]==INF) ?
 INF : (dist[i][k] + dist[k][j]);
                if (dist[i][j] > tmp) {
                    // "i到j最短路径"对应的值设，为更小的一个(即经过k)
                    dist[i][j] = tmp;
                    // "i到j最短路径"对应的路径，经过k
                    path[i][j] = path[i][k];
                }
            }
        }
```

```
    }

    // 打印floyd最短路径的结果
    System.out.printf("floyd: \n");
    for (int i = 0; i < mVexs.length; i++) {
        for (int j = 0; j < mVexs.length; j++)
            System.out.printf("%2d  ", dist[i][j]);
        System.out.printf("\n");
    }
}
```

一、什么是哈希表

哈希表就是一种以 键-值(key-indexed) 存储数据的结构，我们只要输入待查找的值即key，即可查找到其对应的值。

哈希的思路很简单，如果所有的键都是整数，那么就可以使用一个简单的无序数组来实现：将键作为索引，值即为其对应的值，这样就可以快速访问任意键的值。这是对于简单的键的情况，我们将其扩展到可以处理更加复杂的类型的键。

使用哈希查找有两个步骤:

1. 使用哈希函数将被查找的键转换为数组的索引。在理想的情况下，不同的键会被转换为不同的索引值，但是在有些情况下我们需要处理多个键被哈希到同一个索引值的情况。所以哈希查找的第二个步骤就是处理冲突
2. 处理哈希碰撞冲突。有很多处理哈希碰撞冲突的方法，本文后面会介绍拉链法和线性探测法。

哈希表是一个在时间和空间上做出权衡的经典例子。如果没有内存限制，那么可以直接将键作为数组的索引。那么所有的查找时间复杂度为O(1)；如果没有时间限制，那么我们可以使用无序数组并进行顺序查找，这样只需要很少的内存。哈希表使用了适度的时间和空间来在这两个极端之间找到了平衡。只需要调整哈希函数算法即可在时间和空间上做出取舍。

二、哈希函数

哈希查找第一步就是使用哈希函数将键映射成索引。这种映射函数就是哈希函数。如果我们有一个保存0-M数组，那么我们就需要一个能够将任意键转换为该数组范围内的索引（0~M-1）的哈希函数。哈希函数需要易于计算并且能够均匀分布所有键。比如举个简单的例子，使用手机号码后三位就比前三位作为key更好，因为前三位手机号码的重复率很高。再比如使用身份证号码出生年月位数要比使用前几位数要更好。

在实际中，我们的键并不都是数字，有可能是字符串，还有可能是几个值的组合等，所以我们需要实现自己的哈希函数。

1. 正整数

获取正整数哈希值最常用的方法是使用除留余数法。即对于大小为素数M的数组，对于任意正整数k，计算k除以M的余数。M一般取素数。

2. 字符串

将字符串作为键的时候，我们也可以将他作为一个大的整数，采用保留除余法。我们可以将组成字符串的每一个字符取值然后进行哈希，比如

```
public int GetHashCode(string str)
{
    char[] s = str.ToCharArray();
    int hash = 0;
    for (int i = 0; i < s.Length; i++)
    {
        hash = s[i] + (31 * hash);
    }
    return hash;
}
```

上面的哈希值是Horner计算字符串哈希值的方法，公式为:

$$h = s[0] \cdot 31^{L-1} + \ldots + s[L-3] \cdot 31^{2} + s[L-2] \cdot 31^{1} + s[L-1] \cdot 31^{0}$$

举个例子，比如要获取"call"的哈希值，字符串c对应的unicode为99，a对应的unicode为97，L对应的unicode为108，所以字符串"call"的哈希值为

$$3045982 = 99 \cdot 31^{3} + 97 \cdot 31^{2} + 108 \cdot 31^{1} + 108 \cdot 31^{0} = 108 + 31 \cdot (108 + 31 \cdot (97 + 31 \cdot (99)))$$

如果对每个字符去哈希值可能会比较耗时，所以可以通过间隔取N个字符来获取哈希值来节省时间，比如，可以 获取每8-9个字符来获取哈希值：

```
public int GetHashCode(string str)
{
    char[] s = str.ToCharArray();
    int hash = 0;
    int skip = Math.Max(1, s.Length / 8);
    for (int i = 0; i < s.Length; i+=skip)
    {
        hash = s[i] + (31 * hash);
    }
    return hash;
}
```

但是，对于某些情况，不同的字符串会产生相同的哈希值，这就是前面说到的哈希冲突（Hash Collisions），比如下面的四个字符串：

```
http://www.cs.princeton.edu/introcs/13loop/Hello.java
http://www.cs.princeton.edu/introcs/13loop/Hello.class
http://www.cs.princeton.edu/introcs/13loop/Hello.html
http://www.cs.princeton.edu/introcs/12type/index.html
```

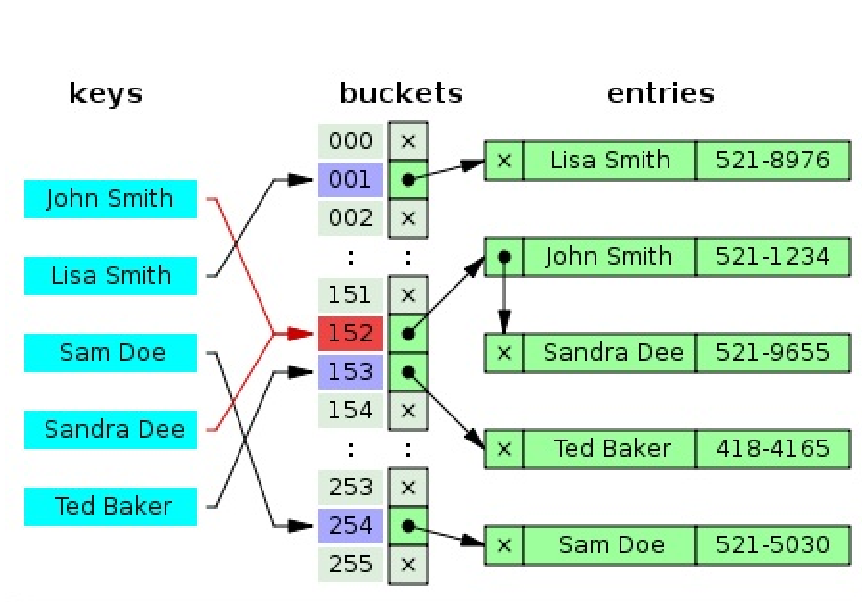
如果我们按照每8个字符取哈希的话，就会得到一样的哈希值。所以下面来讲解如何解决哈希碰撞：

三、避免哈希冲突

拉链法

通过哈希函数，我们可以将键转换为数组的索引(0-M-1)，但是对于两个或者多个键具有相同索引值的情况，我们需要有一种方法来处理这种冲突。

一种比较直接的办法就是，将大小为M 的数组的每一个元素指向一个条链表，链表中的每一个节点都存储散列值为该索引的键值对，这就是拉链法。下图很清楚的描述了什么是拉链法。



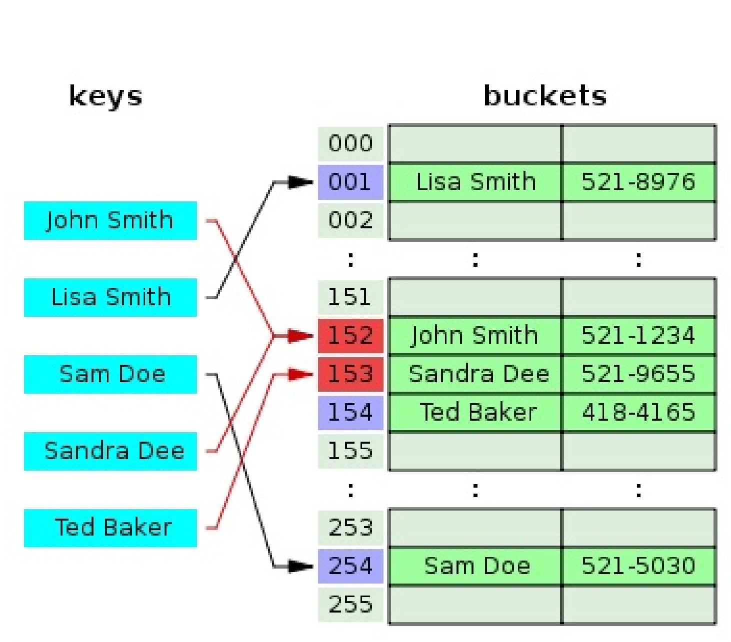
图中，"John Smith"和"Sandra Dee" 通过哈希函数都指向了152 这个索引，该索引又指向了一个链表， 在链表中依次存储了这两个字符串。

该方法的基本思想就是选择足够大的M，使得所有的链表都尽可能的短小，以保证查找的效率。对采用拉链法的哈希实现的查找分为两步，首先是根据散列值找到对应的链表，然后沿着链表顺序找到相应的键。

实现基于拉链表的散列表，目标是选择适当的数组大小M，使得既不会因为空链表而浪费内存空间，也不会因为链表太而在查找上浪费太多时间。拉链表的优点在于，这种数组大小M的选择不是关键性的，如果存入的键多于预期，那么查找的时间只会比选择更大的数组稍长，另外，我们也可以使用更高效的结构来代替链表存储。如果存入的键少于预期，索然有些浪费空间，但是查找速度就会很快。所以当内存不紧张时，我们可以选择足够大的M，可以使得查找时间变为常数，如果内存紧张时，选择尽量大的M仍能够将性能提高M倍。

线性探测法

线性探测法是开放寻址法解决哈希冲突的一种方法，基本原理为，使用大小为M的数组来保存N个键值对，其中M>N，我们需要使用数组中的空位解决碰撞冲突。如下图所示：



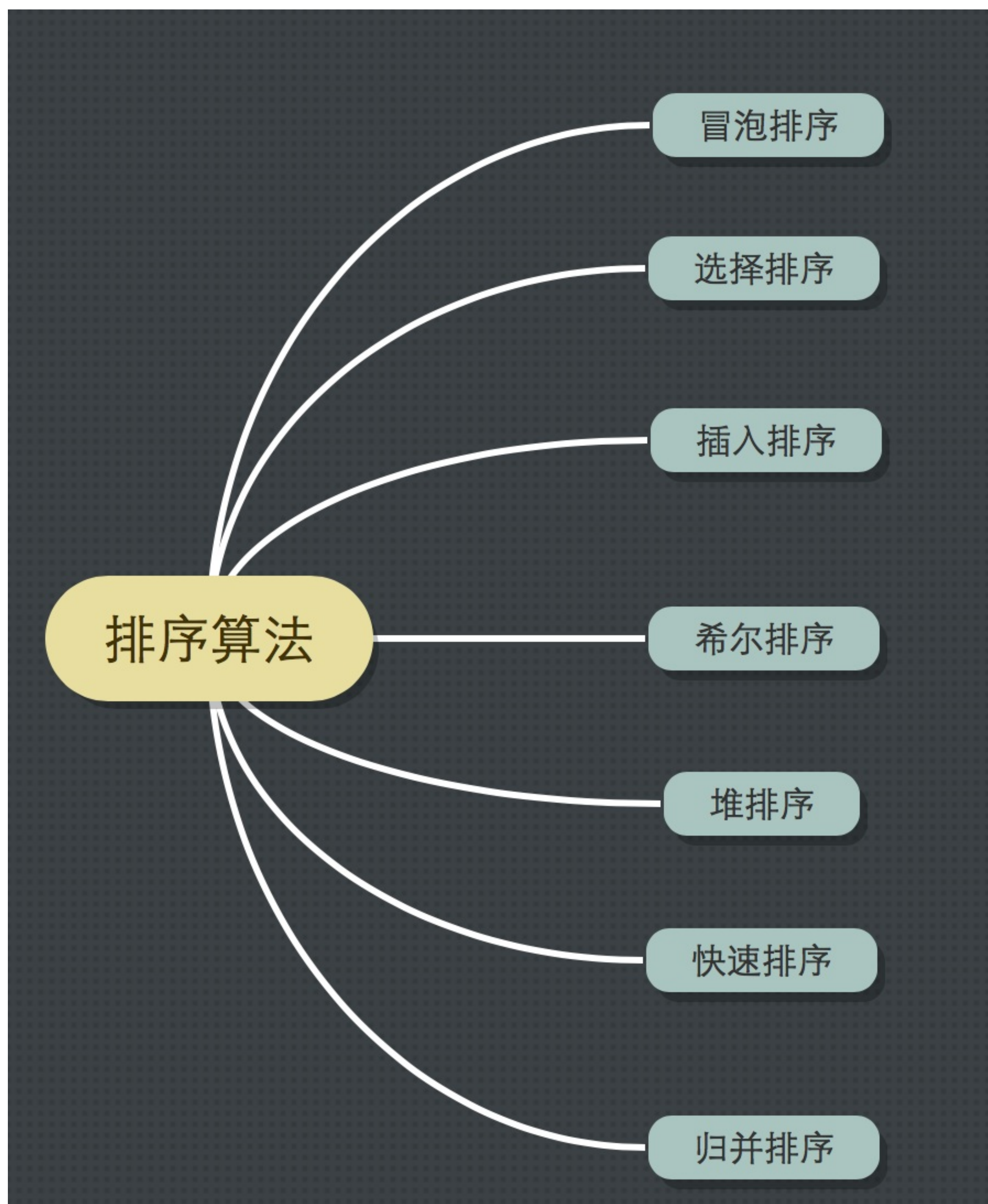
对照前面的拉链法，在该图中，"Ted Baker" 是有唯一的哈希值153的，但是由于153被"Sandra Dee"占用了。而原先"Snadra Dee"和"John Smith"的哈希值都是152的，但是在对"Sandra Dee"进行哈希的时候发现152已经被占用了，所以往下找发现153没有被占用，所以存放在153上，然后"Ted Baker"哈希到153上，发现已经被占用了，所以往下找，发现154没有被占用，所以值存到了154上。

开放寻址法中最简单的是线性探测法：当碰撞发生时即一个键的散列值被另外一个键占用时，直接检查散列表中的下一个位置即将索引值加1，这样的线性探测会出现三种结果：

1. 命中，该位置的键和被查找的键相同
2. 未命中，键为空
3. 继续查找，该位置的键和被查找的键不同。

线性探查（Linear Probing）方式虽然简单，但是有一些问题，它会导致同类哈希的聚集。在存入的时候存在冲突，在查找的时候冲突依然存在。

基本思想:



冒泡排序

基本思想:

比较相邻的元素。如果第一个比第二个大，就交换他们两个。

对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对。在这一点，最后的元素应该会是最大的数。 针对所有的元素重复以上的步骤，除了最后一个。持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。

Java实现

加入标记状态 flag 若在一次冒泡中，没有交换 则说明可以停止 减少运行时

```
public static void bubbleSort(int[] numbers) {
    int temp = 0;
    int size = numbers.length;
    boolean flag = true;
    for (int i = 0; i < size - 1&&flag; i++) {
        flag = false;
        for (int j = 0; j < size - 1 - i; j++) {
            if (numbers[j] > numbers[j + 1]) // 交换两数位置
            {
                temp = numbers[j];
                numbers[j] = numbers[j + 1];
                numbers[j + 1] = temp;
                flag = true;
            }
        }
    }
}
```

时间复杂度O(n*n)

选择排序算法

基本思想：

在要排序的一组数中，选出最小的一个数与第一个位置的数交换；然后在剩下的数当中再找最小的与第二个位置的数交换，如此循环到倒数第二个数和最后一个数比较为止。

Java 实现

```
public static void selectSort(int[] numbers) {
    int size = numbers.length; // 数组长度
    int temp = 0; // 中间变量
    for (int i = 0; i < size-1; i++) {
        int k = i; // 待确定的位置
        // 选择出应该在第i个位置的数
        for (int j = size - 1; j > i; j--) {
            if (numbers[j] < numbers[k]) {
                k = j;
            }
        }
        // 交换两个数
        temp = numbers[i];
        numbers[i] = numbers[k];
        numbers[k] = temp;
    }
}
```

时间复杂度O(n*n) 性能上优于冒泡排序 交换次数少

插入排序算法

基本思想：

每步将一个待排序的记录，按其顺序码大小插入到前面已经排序的字序列的合适位置（从后向前找到合适位置后），直到全部插入排序完为止。

Java 实现

```
public static void insertSort(int[] numbers) {
    int size = numbers.length;
    int temp = 0;
    int j = 0;
    for (int i = 1; i < size; i++) {
        temp = numbers[i];
        // 假如temp比前面的值小，则将前面的值后移
        for (j = i; j > 0 && temp < numbers[j - 1]; j--) {
            numbers[j] = numbers[j - 1];
        }
        numbers[j] = temp;
    }
}
```

时间复杂度

O(n*n) 性能上优于冒泡排序和选择排序

希尔排序算法

基本思想：

先将整个待排序的记录序列分割成为若干子序列分别进行直接插入排序，待整个序列中的记录“基本有序”时，再对全体记录进行依次直接插入排序。

Java 实现

```
/**
 * 希尔排序的原理:根据需求，如果你想要结果从小到大排列，它会首先将数组进行分组，然后将较小值移到前面，较大值
 * 移到后面，最后将整个数组进行插入排序，这样比起一开始就用插入排序减少了数据交换和移动的次数，
 * 可以说希尔排序是加强 版的插入排序 拿数组5, 2,8, 9, 1, 3，4来说，数组长度为7，当increment为3时，数组分为两个序列
 * 5，2，8和9，1，3，4，第一次排序，9和5比较，1和2比较，3和8比较，4和比其下标值小increment的数组值相比较
 * 此例子是按照从小到大排列，所以小的会排在前面，第一次排序后数组为5, 1, 3, 4, 2, 8，9
 * 第一次后increment的值变为3/2=1,此时对数组进行插入排序， 实现数组从大到小排
 */
public static void shellSort(int[] data) {
    int j = 0;
    int temp = 0;
    // 每次将步长缩短为原来的一半
    for (int increment = data.length / 2; increment > 0; increment /= 2) {
        for (int i = increment; i < data.length; i++) {
            temp = data[i];
            for (j = i; j >= increment; j -= increment) {
                if (temp < data[j - increment])// 从小到大排
                {
                    data[j] = data[j - increment];
                } else {
                    break;
                }
            }
            data[j] = temp;
        }
    }
```

时间复杂度O(n^1.5)

堆排序算法

基本思想:

堆排序是一种树形选择排序，是对直接选择排序的有效改进。

堆的定义下：具有n个元素的序列 （h1,h2,...,hn),当且仅当满足（hi>=h2i,hi>=h2i+1）或（hi<=h2i,hi<=h2i+1） (i=1,2,...,n/2)时称之为堆。在这里只讨论满足前者条件的堆。由堆的定义可以看出，堆顶元素（即第一个元素）必为最大项（大顶堆）。完全二叉树可以很直观地表示堆的结构。堆顶为根，其它为左子树、右子树。

思想:初始时把要排序的数的序列看作是一棵顺序存储的二叉树，调整它们的存储序，使之成为一个 堆，这时堆的根节点的数最大。然后将根节点与堆的最后一个节点交换。然后对前面(n-1)个数重新调整使之成为堆。依此类推，直到只有两个节点的堆，并对 它们作交换，最后得到有n个节点的有序序列。从算法描述来看，堆排序需要两个过程，一是建立堆，二是堆顶与堆的最后一个元素交换位置。所以堆排序有两个函数组成。一是建堆的渗透函数，二是反复调用渗透函数实现排序的函数。

Java 实现

```
public static void heapSort(int[] a){
    int arrayLength = a.length;
    // 循环建堆
    for (int i = 0; i < arrayLength - 1; i++) {
        // 建堆
        buildMaxHeap(a, arrayLength - 1 - i);
        // 交换堆顶和最后一个元素
        swap(a, 0, arrayLength - 1 - i);
        System.out.println(Arrays.toString(a));
    }
}
// 对data数组从0到lastIndex建大顶堆
public static void buildMaxHeap(int[] data, int lastIndex) {
    // 从lastIndex处节点（最后一个节点）的父节点开始
    for (int i = (lastIndex - 1) / 2; i >= 0; i--) {
        // k保存正在判断的节点
        int k = i;
        // 如果当前k节点的子节点存在
        while (k * 2 + 1 <= lastIndex) {
```

```
            // k节点的左子节点的索引
            int biggerIndex = 2 * k + 1;
            // 如果biggerIndex小于lastIndex，即biggerIndex+1代表的k节点的右子节点存在
            if (biggerIndex < lastIndex) {
                // 若果右子节点的值较大
                if (data[biggerIndex] < data[biggerIndex + 1]) {
                    // biggerIndex总是记录较大子节点的索引
                    biggerIndex++;
                }
            }
            // 如果k节点的值小于其较大的子节点的值
            if (data[k] < data[biggerIndex]) {
                // 交换他们
                swap(data, k, biggerIndex);
                // 将biggerIndex赋予k，开始while循环的下一次循环，重新保证k节点的值大于其左右子节点的值
                k = biggerIndex;
            } else {
                break;
            }
        }
    }
}
// 交换
private static void swap(int[] data, int i, int j) {
    int tmp = data[i];
    data[i] = data[j];
    data[j] = tmp;
}
```

时间复杂度O(nlogn）不适合待排序序列较少的情况

快速排序算法

基本思想：

通过一趟排序将待排序记录分割成独立的两部分，其中一部分记录的关键字均比另一部分关键字小，则分别对这两部分继续进行排序，直到整个序列有序。

Java 实现

```
/**
 * 快速排序
 *
 * @param numbers
 *            带排序数组
 */
public static void quick(int[] numbers) {
    if (numbers.length > 0) // 查看数组是否为空
    {
        quickSort(numbers, 0, numbers.length - 1);
    }
}
/**
 *
 * @param numbers
 *            带排序数组
 * @param low
 *            开始位置
 * @param high
 *            结束位置
 */
public static void quickSort(int[] numbers, int low, int high) {
    if (low >= high) {
        return;
    }
    int middle = getMiddle(numbers, low, high); // 将numbers数组
进行一分为二
    quickSort(numbers, low, middle - 1); // 对低字段表进行递归排序
    quickSort(numbers, middle + 1, high); // 对高字段表进行递归排序
}
/**
 * 查找出中轴（默认是最低位low）的在numbers数组排序后所在位置
 *
 * @param numbers
 *            带查找数组
```

```
 * @param low
 *            开始位置
 * @param high
 *            结束位置
 * @return 中轴所在位置
 */
public static int getMiddle(int[] numbers, int low, int high) {
    int temp = numbers[low]; // 数组的第一个作为中轴
    while (low < high) {
        while (low < high && numbers[high] > temp) {
            high--;
        }
        numbers[low] = numbers[high];// 比中轴小的记录移到低端
        while (low < high && numbers[low] < temp) {
            low++;
        }
        numbers[high] = numbers[low]; // 比中轴大的记录移到高端
    }
    numbers[low] = temp; // 中轴记录到尾
    return low; // 返回中轴的位置
}
```

时间复杂度O(nlogn）

快速排序在序列中元素很少时，效率将比较低，不如插入排序，因此一般在序列中元素很少时使用插入排序，这样可以提高整体效率。

归并排序算法

基本思想:

归并（Merge）排序法是将两个（或两个以上）有序表合并成一个新的有序表，即把待排序序列分为若干个子序列，每个子序列是有序的。然后再把有序子序列合并为整体有序序列。

Java 实现

```
/**
```

```
 * 归并排序
 * 简介:将两个（或两个以上）有序表合并成一个新的有序表 即把待排序序列分为若干个子序列，每个子序列是有序的。然后再把有序子序列合并为整体有序序列
 * 时间复杂度为O(nlogn)
 * 稳定排序方式
 * @param nums 待排序数组
 * @return 输出有序数组
 */
public static int[] sort(int[] nums, int low, int high) {
    int mid = (low + high) / 2;
    if (low < high) {
        // 左边
        sort(nums, low, mid);
        // 右边
        sort(nums, mid + 1, high);
        // 左右归并
        merge(nums, low, mid, high);
    }
    return nums;
}
/**
 * 将数组中low到high位置的数进行排序
 * @param nums 待排序数组
 * @param low 待排的开始位置
 * @param mid 待排中间位置
 * @param high 待排结束位置
 */
public static void merge(int[] nums, int low, int mid, int high)
{
    int[] temp = new int[high - low + 1];
    int i = low;// 左指针
    int j = mid + 1;// 右指针
    int k = 0;
    // 把较小的数先移到新数组中
    while (i <= mid && j <= high) {
        if (nums[i] < nums[j]) {
            temp[k++] = nums[i++];
        } else {
            temp[k++] = nums[j++];
        }
```

```
        }
        // 把左边剩余的数移入数组
        while (i <= mid) {
            temp[k++] = nums[i++];
        }
        // 把右边边剩余的数移入数组
        while (j <= high) {
            temp[k++] = nums[j++];
        }
        // 把新数组中的数覆盖nums数组
        for (int k2 = 0; k2 < temp.length; k2++) {
            nums[k2 + low] = temp[k2];
        }
    }
```

时间复杂度O(nlogn）

各种算法的时间复杂度等性能比较

各种常用排序算法						
类别	排序方法	时间复杂度			空间复杂度	稳定性
		平均情况	最好情况	最坏情况	辅助存储	
插入排序	直接插入	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	shell排序	$O(n^{1.3})$	$O(n)$	$O(n^2)$	$O(1)$	不稳定
选择排序	直接选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
	堆排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(1)$	不稳定
交换排序	冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	快速排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(n^2)$	$O(nlog_2n)$	不稳定
归并排序		$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(1)$	稳定

一、海量数据处理

所谓海量数据处理，无非就是基于海量数据上的存储、处理、操作。何谓海量，就是数据量太大，所以导致要么是无法在较短时间内迅速解决，要么是数据太大，导致无法一次性装入内存。

那解决办法呢?

针对时间，我们可以采用巧妙的算法搭配合适的数据结构，如Bloom filter/Hash/bit-map/堆/trie树。

针对空间，无非就一个办法：大而化小，分而治之（hash映射）。

二、算法/数据结构基础

1.Bloom Filter

Bloom Filter（BF）是一种空间效率很高的随机数据结构，它利用位数组很简洁地表示一个集合，并能判断一个元素是否属于这个集合。它是一个判断元素是否存在集合的快速的概率算法。Bloom Filter有可能会出现错误判断，但不会漏掉判断。也就是Bloom Filter判断元素不再集合，那肯定不在。如果判断元素存在集合中，有一定的概率判断错误。因此，Bloom Filter不适合那些“零错误”的应用场合。

而在能容忍低错误率的应用场合下，Bloom Filter比其他常见的算法（如hash，折半查找）极大节省了空间。

适用范围

可以用来实现数据字典，进行数据的判重，或者集合求交集

具体参考：海量数据处理之Bloom Filter详解

2.Hash

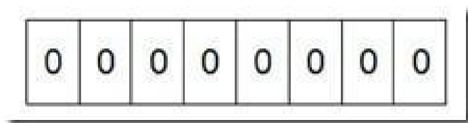
Hash，一般翻译做“散列”，也有直接音译为“哈希”的，就是把任意长度的输入（又叫做预映射， **pre-image**），通过散列算法，变换成固定长度的输出，该输出就是散列值。这种转换是一种压缩映射，也就是，散列值的空间通常远小于输入的空间，不同的输入可能会散列成相同的输出，而不可能从散列值来唯一的确定输入值。简单的说就是一种将任意长度的消息压缩到某一固定长度的消息摘要的函数。

具体参考：从头到尾解析Hash表算法

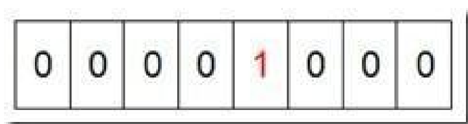
3. Bit-map

所谓的Bit-map就是用一个bit位来标记某个元素对应的值。由于采用了Bit为单位来存储数据，因此在存储空间方面，可以大大节省。

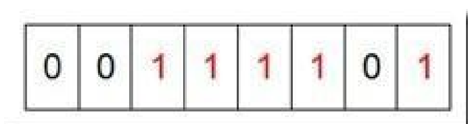
如果说了这么多还没明白什么是Bit-map，那么我们来看一个具体的例子，假设我们要对0-7内的5个元素(4,7,2,5,3)排序（这里假设这些元素没有重复）。那么我们就可以采用Bit-map的方法来达到排序的目的。要表示8个数，我们就只需要8个Bit（1Bytes），首先我们开辟1Byte的空间，将这些空间的所有Bit位都置为0(如下图：)



然后遍历这5个元素，首先第一个元素是4，那么就把4对应的位置为1（可以这样操作 p+(i/8)|(0x01<<(i%8)) 当然了这里的操作涉及到Big-ending和Little-ending的情况，这里默认为Big-ending）,因为是从零开始的，所以要把第五位置为一（如下图）：



然后再处理第二个元素7，将第八位置为1,，接着再处理第三个元素，一直到最后处理完所有的元素，将相应的位置为1，这时候的内存的Bit位的状态如下：



具体参考：数据结构：位图法

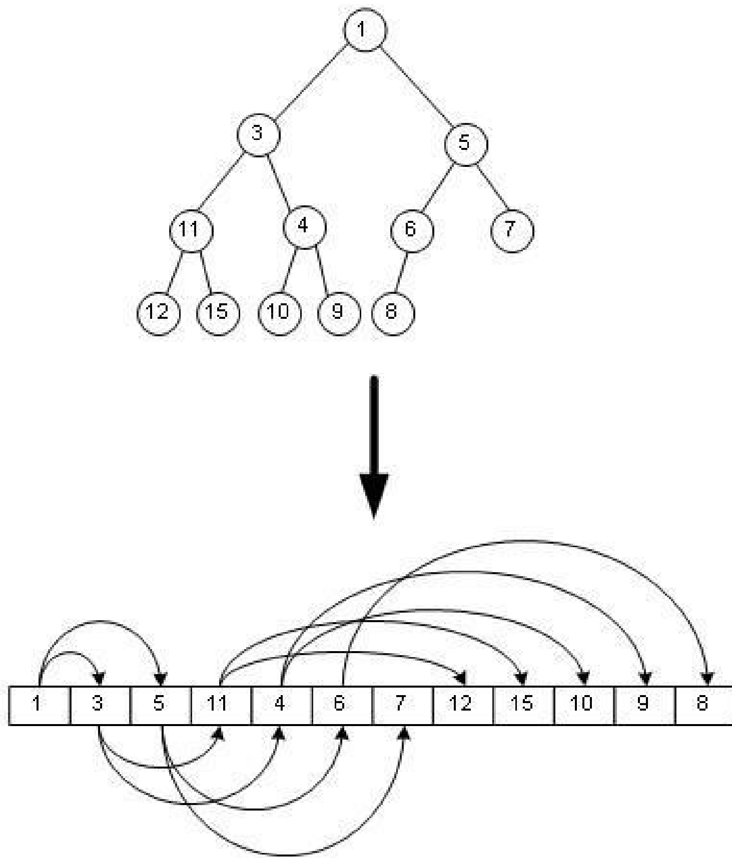
4.堆

堆是一种特殊的二叉树，具备以下两种性质

1）每个节点的值都大于（或者都小于，称为最小堆）其子节点的值

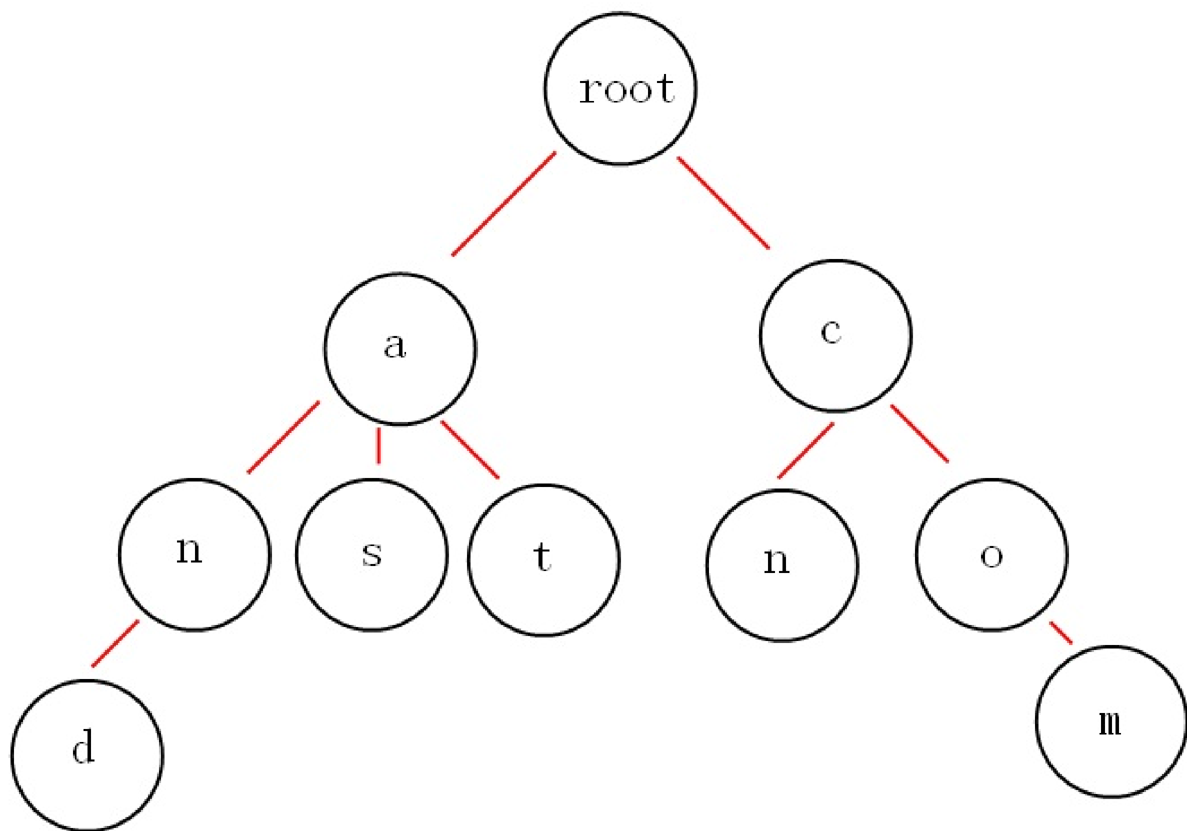
2）树是完全平衡的，并且最后一层的树叶都在最左边这样就定义了一个最大堆。

如下图用一个数组来表示堆：



5.trie树

下面我们有and,as,at,cn,com这些关键词，那么如何构建trie树呢？



从上面的图中，我们或多或少的可以发现一些好玩的特性。

第一：根节点不包含字符，除根节点外的每一个子节点都包含一个字符。

第二：从根节点到某一节点，路径上经过的字符连接起来，就是该节点对应的字符串。

第三：每个单词的公共前缀作为一个字符节点保存。

适用范围：

前缀统计，词频统计。

具体参考：6天通吃树结构—— 第五天 Trie树

6.外排序

适用范围：

大数据的排序，去重

基本原理及要点：

外部排序的两个独立阶段：

1）首先按内存大小，将外存上含n个记录的文件分成若干长度L的子文件或段。依次读入内存并利用有效的内部排序对他们进行排序，并将排序后得到的有序字文件重新写入外存，通常称这些子文件为归并段。

2）对这些归并段进行逐趟归并，使归并段逐渐由小到大，直至得到整个有序文件为之。

外排序的优化方法：置换选择 败者树原理，最优归并树

具体参考：选择置换+败者树搞定外部排序

三、面试问题解决

①、海量日志数据，提取出某日访问百度次数最多的那个IP。

算法思想：分而治之**+Hash**

1.IP地址最多有2^32=4G种取值情况，所以不能完全加载到内存中处理；

2.可以考虑采用“分而治之”的思想，按照IP地址的Hash(IP)%1024值，把海量IP日志分别存储到1024个小文件中。这样，每个小文件最多包含4MB个IP地址；

3.对于每一个小文件，可以构建一个IP为key，出现次数为value的Hash map，同时记录当前出现次数最多的那个IP地址；

4.可以得到1024个小文件中的出现次数最多的IP，再依据常规的排序算法得到总体上出现次数最多的IP；

②、搜索引擎会通过日志文件把用户每次检索使用的所有检索串都记录下来，每个查询串的长度为1-255字节。假设目前有一千万个记录（这些查询串的重复度比较高，虽然总数是1千万，但如果除去重复后，不超过3百万个。一个查询串的重复度越高，说明查询它的用户越多，也就是越热门。），请你统计最热门的10个查询串，要求使用的内存不能超过1G。

可以在内存中处理，典型的**Top K**算法

算法思想：**hashmap+**堆

1.先对这批海量数据预处理，在O（N）的时间内用Hash表完成统计；

2.借助堆这个数据结构，找出Top K，时间复杂度为O(N*logK)。

或者：采用trie树，关键字域存该查询串出现的次数，没有出现为0。最后用10个元素的最小推来对出现频率进行排序。

③、有一个1G大小的一个文件，里面每一行是一个词，词的大小不超过16字节，内存限制大小是1M。返回频数最高的100个词。

算法思想：分而治之 + hash统计 + 堆排序

1.顺序读文件中，对于每个词x，取hash(x)%5000，然后按照该值存到5000个小文件（记为x0,x1,...x4999）中。这样每个文件大概是200k左右。如果其中的有的文件超过了1M大小，还可以按照类似的方法继续往下分，直到分解得到的小文件的大小都不超过1M。

2.对每个小文件，采用trie树/hash_map等统计每个文件中出现的词以及相应的频率。

3.取出出现频率最大的100个词（可以用含100个结点的最小堆）后，再把100个词及相应的频率存入文件，这样又得到了5000个文件。最后就是把这5000个文件进行归并（类似于归并排序）的过程了。

④、有10个文件，每个文件1G，每个文件的每一行存放的都是用户的query，每个文件的query都可能重复。要求你按照query的频度排序。

方案**1**：

算法思想：分而治之 + hash统计 + 堆排序

顺序读取10个文件，按照hash(query)%10的结果将query写入到另外10个文件中。这样新生成的文件每个的大小大约也1G，大于1G继续按照上述思路分。

找一台内存在2G左右的机器，依次对用hash_map(query, query_count)来统计每个query出现的次数。利用快速/堆/归并排序按照出现次数进行排序。将排序好的query和对应的query_cout输出到文件中。这样得到了10个排好序的文件（记为）。

对这10个文件进行归并排序（内排序与外排序相结合）。

方案**2**：

算法思想：**hashmap+**堆

一般query的总量是有限的，只是重复的次数比较多而已，可能对于所有的query，一次性就可以加入到内存了。这样，我们就可以采用trie树/hash_map等直接来统计每个query出现的次数，然后按出现次数做快速/堆/归并排序就可以了。

⑤、 给定a、b两个文件，各存放50亿个url，每个url各占64字节，内存限制是4G，让你找出a、b文件共同的url

方案**1**：可以估计每个文件安的大小为5G×64=320G，远远大于内存限制的4G。所以不可能将其完全加载到内存中处理。考虑采取分而治之的方法。

算法思想：分而治之 **+ hash**统计

遍历文件a，对每个url求取hash(url)%1000，然后根据所取得的值将url分别存储到1000个小文件（记为a0,a1,...,a999）中。这样每个小文件的大约为300M。

遍历文件b，采取和a相同的方式将url分别存储到1000小文件（记为b0,b1,...,b999）。这样处理后，所有可能相同的url都在对应的小文件（a0vsb0,a1vsb1,...,a999vsb999）中，不对应的小文件不可能有相同的url。然后我们只要求出1000对小文件中相同的url即可。

求每对小文件中相同的url时，可以把其中一个小文件的url存储到hash_set中。然后遍历另一个小文件的每个url，看其是否在刚才构建的hash_set中，如果是，那么就是共同的url，存到文件里面就可以了。

方案**2**：如果允许有一定的错误率，可以使用Bloom filter，4G内存大概可以表示340亿bit。将其中一个文件中的url使用Bloom filter映射为这340亿bit，然后挨个读取另外一个文件的url，检查是否与Bloom filter，如果是，那么该url应该是共同的url（注意会有一定的错误率）。

⑥、在2.5亿个整数中找出不重复的整数，注，内存不足以容纳这2.5亿个整数。

采用2-Bitmap（每个数分配2bit，00表示不存在，01表示出现一次，10表示多次，11无意义）进行，共需内存2^32 * 2 bit=1 GB内存，还可以接受。然后扫描这2.5亿个整数，查看Bitmap中相对应位，如果是00变01，01变10，10保持不变。所描完事后，查看bitmap，把对应位是01的整数输出即可。

⑦、给40亿个不重复的unsigned int的整数，没排过序的，然后再给一个数，如何快速判断这个数是否在那40亿个数当中？

方案**1**：申请512M的内存，一个bit位代表一个unsigned int值。读入40亿个数，设置相应的bit位，读入要查询的数，查看相应bit位是否为1，为1表示存在，为0表示不存在。

方案**2**：因为2^32为40亿多，所以给定一个数可能在，也可能不在其中；

这里我们把40亿个数中的每一个用32位的二进制来表示

假设这40亿个数开始放在一个文件中。

然后将这40亿个数分成两类:

1.最高位为0

2.最高位为1

并将这两类分别写入到两个文件中，其中一个文件中数的个数<=20亿，而另一个>=20亿（这相当于折半了）；

与要查找的数的最高位比较并接着进入相应的文件再查找

再然后把这个文件为又分成两类:

1.次最高位为0

2.次最高位为1

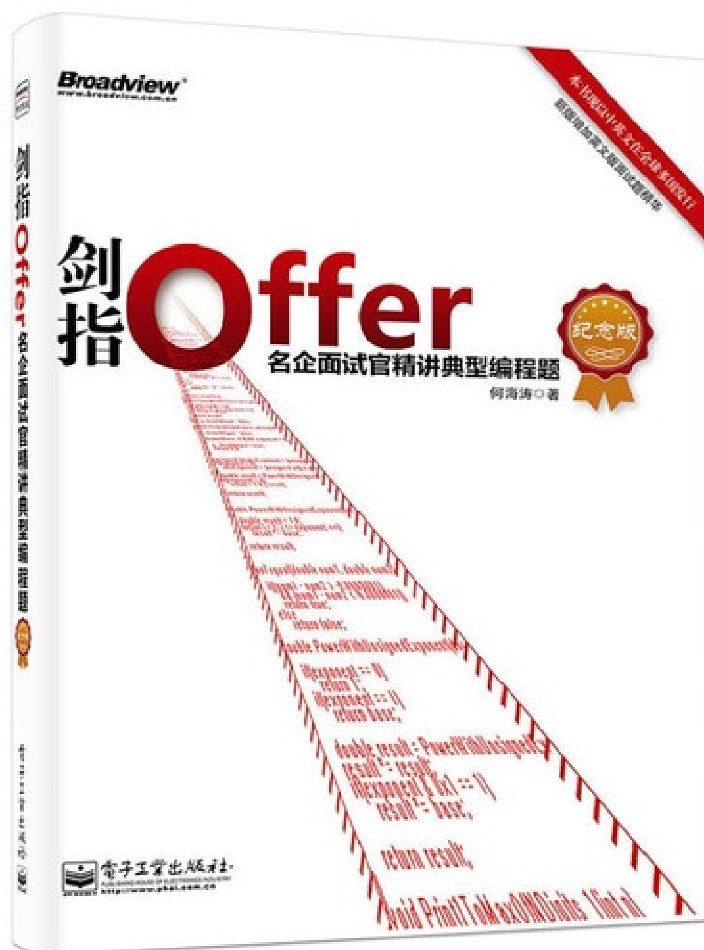
并将这两类分别写入到两个文件中，其中一个文件中数的个数<=10亿，而另一个>=10亿（这相当于折半了）；

与要查找的数的次最高位比较并接着进入相应的文件再查找。

....

以此类推，就可以找到了。

一、前言



剑指offer这本书的重要性不言而喻，题目不是很难，主要考察一些基本的算法思路及数据结构。其中很多题目更在面试中高频出现。

本部分内容整理了剑指offer中的所有题目，提供了详细的解题思路及Java代码实现，希望能对大家的面试有帮助！

二、目录

一、题目

在一个二维数组中，每一行都按照从左到右递增的顺序排序，每一列都按照从上到下递增的顺序排序。请完成一个函数，输入这样的一个二维数组和一个整数，判断数组中是否含有该整数。

二、解题思路

首先选取数组中右上角的数字。如果该数字等于要查找的数字，查找过程结束。

如果该数字大于要查找的数字，剔除这个数字所在的列：如果该数字小于要查找的数字，剔除这个数字所在的行。

也就是说如果要查找的数字不在数组的右上角，则每一次都在数组的查找范围中剔除行或者一列，这样每一步都可以缩小查找的范围，直到找到要查找的数字，或者查找范围为空。

三、解题代码

```
public class Test {
    public static boolean find(int[][] matrix, int number) {

        // 输入条件判断
        if (matrix == null || matrix.length < 1 || matrix[0].length < 1) {
            return false;
        }

        int rows = matrix.length; // 数组的行数
        int cols = matrix[1].length; // 数组行的列数

        int row = 0; // 起始开始的行号
        int col = cols - 1; // 起始开始的列号

        // 要查找的位置确保在数组之内
        while (row >= 0 && row < rows && col >= 0 && col < cols) {
            if (matrix[row][col] == number) { // 如果找到了就直接退出
                return true;
            } else if (matrix[row][col] > number) { // 如果找到的数比要找的数大，说明要找的数在当前数的左边
                col--; // 列数减一，代表向左移动
            } else { // 如果找到的数比要找的数小，说明要找的数在当前数的下边
                row++; // 行数加一，代表向下移动
            }
        }

        return false;
    }
}
```

一、题目

请实现一个函数，把字符串中的每个空格替换成"%20"，例如“We are happy.”，则输出“We%20are%20happy.”。

二、解题思路

先判断字符串中空格的数量。根据数量判断该字符串有没有足够的空间替换成"%20"。

如果有足够空间，计算出需要的空间。根据最终需要的总空间，维护一个指针在最后。从后到前，遇到非空的就把该值挪到指针指向的位置，然后指针向前一位，遇到“ ”，则指针前移，依次替换为“02%”。

三、解题代码

```
public class Test {
    /**
     * 请实现一个函数，把字符串中的每个空格替换成"%20"，例如“We are happ
y.“，则输出”We%20are%20happy.“。
     *
     * @param string     要转换的字符数组
     * @param usedLength 已经字符数组中已经使用的长度
     * @return 转换后使用的字符长度，-1表示处理失败
     */
    public static int replaceBlank(char[] string, int usedLength)
 {
        // 判断输入是否合法
        if (string == null || string.length < usedLength) {
            return -1;
        }

        // 统计字符数组中的空白字符数
        int whiteCount = 0;
        for (int i = 0; i < usedLength; i++) {
            if (string[i] == ' ') {
```

```
                whiteCount++;
            }
        }

        // 计算转换后的字符长度是多少
        int targetLength = whiteCount * 2 + usedLength;
        int tmp = targetLength; // 保存长度结果用于返回
        if (targetLength > string.length) { // 如果转换后的长度大于
数组的最大长度，直接返回失败
            return -1;
        }

        // 如果没有空白字符就不用处理
        if (whiteCount == 0) {
            return usedLength;
        }

        usedLength--; // 从后向前，第一个开始处理的字符
        targetLength--; // 处理后的字符放置的位置

        // 字符中有空白字符，一直处理到所有的空白字符处理完
        while (usedLength >= 0 && usedLength < targetLength) {
            // 如是当前字符是空白字符，进行"%20"替换
            if (string[usedLength] == ' ') {
                string[targetLength--] = '0';
                string[targetLength--] = '2';
                string[targetLength--] = '%';
            } else { // 否则移动字符
                string[targetLength--] = string[usedLength];
            }
            usedLength--;
        }

        return tmp;
    }
}
```

一、题目

输入个链表的头结点，从尾到头反过来打印出每个结点的值。

二、解题思路

使用栈的方式进行。

将链表从头到尾压入栈内，出栈的过程就对应着从尾到头。

三、解题代码

```
public class Test {
    /**
     * 结点对象
     */
    public static class ListNode {
        int val; // 结点的值
        ListNode nxt; // 下一个结点
    }

    /**
     * 输入个链表的头结点，从尾到头反过来打印出每个结点的值
     * 使用栈的方式进行
     *
     * @param root 链表头结点
     */
    public static void printListInverselyUsingIteration(ListNode
 root) {
        Stack<ListNode> stack = new Stack<>();
        while (root != null) {
            stack.push(root);
            root = root.nxt;
        }
        ListNode tmp;
        while (!stack.isEmpty()) {
            tmp = stack.pop();
```

```
            System.out.print(tmp.val + " ");
        }
    }

    /**
     * 输入个链表的头结点，从尾到头反过来打印出每个结点的值
     * 使用递归的方式进行
     *
     * @param root 链表头结点
     */
    public static void printListInverselyUsingRecursion(ListNode
 root) {
        if (root != null) {
            printListInverselyUsingRecursion(root.nxt);
            System.out.print(root.val + " ");
        }
    }
}
```

一、题目

输入某二叉树的前序遍历和中序遍历的结果，请重建出该二叉树。假设输入的前序遍历和中序遍历的结果中都不含重复的数字。例如：前序遍历序列｛ 1, 2, 4, 7, 3, 5, 6, 8｝和中序遍历序列｛4, 7, 2, 1, 5, 3, 8，6}，重建二叉树并输出它的头结点。

二、解题思路

由前序遍历的第一个节点可知根节点。根据根节点，可以将中序遍历划分成左右子树。在前序遍历中找出对应的左右子树，其第一个节点便是根节点的左右子节点。按照上述方式递归便可重建二叉树。

三、解题代码

```
public class Test {
    /**
     * 二叉树节点类
     */
    public static class BinaryTreeNode {
        int value;
        BinaryTreeNode left;
        BinaryTreeNode right;
    }

    /**
     * 输入某二叉树的前序遍历和中序遍历的结果，请重建出该二节树。假设输入的
前序遍历和中序遍历的结果中都不含重复的数字。
     *
     * @param preorder 前序遍历
     * @param inorder  中序遍历
     * @return 树的根结点
     */
    public static BinaryTreeNode construct(int[] preorder, int[]
 inorder) {
        // 输入的合法性判断，两个数组都不能为空，并且都有数据，而且数据的数
目相同
```

```
        if (preorder == null || inorder == null || preorder.leng
th != inorder.length || inorder.length < 1) {
            return null;
        }

        return construct(preorder, 0, preorder.length - 1, inord
er, 0, inorder.length - 1);
    }

    /**
     * 输入某二叉树的前序遍历和中序遍历的结果，请重建出该二节树。假设输入的
前序遍历和中序遍历的结果中都不含重复的数字。
     *
     * @param preorder 前序遍历
     * @param ps       前序遍历的开始位置
     * @param pe       前序遍历的结束位置
     * @param inorder  中序遍历
     * @param is       中序遍历的开始位置
     * @param ie       中序遍历的结束位置
     * @return 树的根结点
     */
    public static BinaryTreeNode construct(int[] preorder, int p
s, int pe, int[] inorder, int is, int ie) {

        // 开始位置大于结束位置说明已经没有需要处理的元素了
        if (ps > pe) {
            return null;
        }
        // 取前序遍历的第一个数字，就是当前的根结点
        int value = preorder[ps];
        int index = is;
        // 在中序遍历的数组中找根结点的位置
        while (index <= ie && inorder[index] != value) {
            index++;
        }

        // 如果在整个中序遍历的数组中没有找到，说明输入的参数是不合法的，抛
出异常
        if (index > ie) {
            throw new RuntimeException("Invalid input");
```

```
        }

        // 创建当前的根结点，并且为结点赋值
        BinaryTreeNode node = new BinaryTreeNode();
        node.value = value;

        // 递归构建当前根结点的左子树，左子树的元素个数：index-is+1个
        // 左子树对应的前序遍历的位置在[ps+1, ps+index-is]
        // 左子树对应的中序遍历的位置在[is, index-1]
        node.left = construct(preorder, ps + 1, ps + index - is,
 inorder, is, index - 1);
        // 递归构建当前根结点的右子树，右子树的元素个数：ie-index个
        // 右子树对应的前序遍历的位置在[ps+index-is+1, pe]
        // 右子树对应的中序遍历的位置在[index+1, ie]
        node.right = construct(preorder, ps + index - is + 1, pe
, inorder, index + 1, ie);

        // 返回创建的根结点
        return node;
    }
}
```

一、题目

用两个栈实现一个队列。队列的声明如下，请实现它的两个函数appendTail 和deleteHead，分别完成在队列尾部插入结点和在队列头部删除结点的功能。

二、解题思路

栈1用于存储元素，栈2用于弹出元素，负负得正。

说的通俗一点，现在把数据1、2、3分别入栈一，然后从栈一中出来（3、2、1），放到栈二中，那么，从栈二中出来的数据（1、2、3）就符合队列的规律了，即负负得正。

三、解题代码

```
public class Test {
    public static class MList<T> {
        // 插入栈，只用于插入的数据
        private Stack<T> stack1 = new Stack<>();
        // 弹出栈，只用于弹出数据
        private Stack<T> stack2 = new Stack<>();

        public MList() {
        }

        // 添加操作，成在队列尾部插入结点
        public void appendTail(T t) {
            stack1.add(t);
        }

        // 删除操作，在队列头部删除结点
        public T deleteHead() {

            // 先判断弹出栈是否为空，如果为空就将插入栈的所有数据弹出栈，

            // 并且将弹出的数据压入弹出栈中
            if (stack2.isEmpty()) {
                while (!stack1.isEmpty()) {
                    stack2.add(stack1.pop());
                }
            }

            // 如果弹出栈中还没有数据就抛出异常
            if (stack2.isEmpty()) {
                throw new RuntimeException("No more element.");

            }

            // 返回弹出栈的栈顶元素，对应的就是队首元素。
            return stack2.pop();
        }
    }
}
```

一、题目

把一个数组最开始的若干个元素搬到数组的末尾， 我们称之数组的旋转。输入一个递增排序的数组的一个旋转， 输出旋转数组的最小元素。例如数组{3,4,5,1,2｝为｛1,2,3,4,5}的一个旋转，该数组的最小值为1。

二、解题思路

Step1.和二分查找法一样，我们用两个指针分别指向数组的第一个元素和最后一个元素。

Step2.接着我们可以找到数组中间的元素：

如果该中间元素位于前面的递增子数组，那么它应该大于或者等于第一个指针指向的元素。此时数组中最小的元素应该位于该中间元素的后面。我们可以把第一个指针指向该中间元素，这样可以缩小寻找的范围。如果中间元素位于后面的递增子数组，那么它应该小于或者等于第二个指针指向的元素。此时该数组中最小的元素应该位于该中间元素的前面。

Step3.接下来我们再用更新之后的两个指针，重复做新一轮的查找。

三、解题代码

```
public class Test {

    /**
     * @param numbers 旋转数组
     * @return 数组的最小值
     */
    public static int min(int[] numbers) {
        // 判断输入是否合法
        if (numbers == null || numbers.length == 0) {
            throw new RuntimeException("Invalid input.");
        }

        // 开始处理的第一个位置
```

```
        int lo = 0;
        // 开始处理的最后一个位置
        int hi = numbers.length - 1;
        // 设置初始值
        int mi = lo;

        // 确保lo在前一个排好序的部分，hi在排好序的后一个部分
        while (numbers[lo] >= numbers[hi]) {
            // 当处理范围只有两个数据时，返回后一个结果
            // 因为numbers[lo] >= numbers[hi]总是成立，后一个结果对应的是最小的值
            if (hi - lo == 1) {
                return numbers[hi];
            }

            // 取中间的位置
            mi = lo + (hi - lo) / 2;

            // 如果三个数都相等，则需要进行顺序处理，从头到尾找最小的值
            if (numbers[mi] == numbers[lo] && numbers[hi] == numbers[mi]) {
                return minInorder(numbers, lo, hi);
            }

            // 如果中间位置对应的值在前一个排好序的部分，将lo设置为新的处理位置
            if (numbers[mi] >= numbers[lo]) {
                lo = mi;
            }
            // 如果中间位置对应的值在后一个排好序的部分，将hi设置为新的处理位置
            else if (numbers[mi] <= numbers[hi]) {
                hi = mi;
            }
        }

        // 返回最终的处理结果
        return numbers[mi];
    }
```

```
    /**
     * 找数组中的最小值
     *
     * @param numbers 数组
     * @param start   数组的起始位置
     * @param end     数组的结束位置
     * @return 找到的最小的数
     */
    public static int minInorder(int[] numbers, int start, int e
nd) {
        int result = numbers[start];
        for (int i = start + 1; i <= end; i++) {
            if (result > numbers[i]) {
                result = numbers[i];
            }
        }
        return result;
    }
}
```

一、题目

写一个函数，输入n，求斐波那契数列的第n项值。

斐波那契数列的定义如下：

$$F_0 = 0$$

$$F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2}$$

二、解题思路

按照上述递推式，可以使用循环或递归的方式获取第n项式。

三、解题代码

```
public class Test {

    /**
     * 写一个函数，输入n，求斐波那契（Fibonacci) 数列的第n项
     * @param n Fibonacci数的项数
     * @return 第n项的结果
     */
    public static long fibonacci(int n) {

        // 当输入非正整数的时候返回0
        if (n <= 0) {
            return 0;
        }

        // 输入1或者2的时候返回1
        if (n == 1 || n == 2) {
            return 1;
        }

        // 第n-2个的Fibonacci数的值
        long prePre = 1;
        // 第n-1个的Fibonacci数的值
        long pre = 1;
        // 第n个的Fibonacci数的值
        long current = 2;

        // 求解第n个的Fibonacci数的值
        for (int i = 3; i <= n ; i++) {
            // 求第i个的Fibonacci数的值
            current = prePre + pre;
            // 更新记录的结果，prePre原先记录第i-2个Fibonacci数的值
            // 现在记录第i-1个Fibonacci数的值
            prePre = pre;
            // 更新记录的结果，pre原先记录第i-1个Fibonacci数的值
            // 现在记录第i个Fibonacci数的值
            pre = current;
        }

        // 返回所求的结果
```

```
            return current;
        }
    }
```

一、题目

请实现一个函数，输入一个整数，输出该数二进制表示中1的个数。例如把9表示成二进制1001，有2位1。因此如果输入9，该函数输出2。

二、解题思路

①位移＋计数 每次右移一位，不断和1进行与运算，直到位0。

②循环让(n - 1) & n。如果n的二进制表示中有k个1，那么这个方法只需要循环k次即可。其原理是不断清除**n**的二进制表示中最右边的**1**，同时累加计数器，直至**n**为**0**。因为从二进制的角度讲，n相当于在n - 1的最低位加上1。举个例子，8（1000）= 7（0111）+ 1（0001），所以8 & 7 = （1000）&（0111）= 0（0000），清除了8最右边的1（其实就是最高位的1，因为8的二进制中只有一个1）。再比如7（0111）= 6（0110）+ 1（0001），所以7 & 6 = （0111）&（0110）= 6（0110），清除了7的二进制表示中最右边的1（也就是最低位的1）。

三、解题代码

```
public class Test {

    /**
     * 请实现一个函数， 输入一个整数，输出该数二进制表示中1的个数。
     * 例如把9表示成二进制是1001 ，有2位是1. 因此如果输入9，该出2。
     *
     * @param n 待的数字
     * @return 数字中二进制表表的1的数目
     */
    public static int numberOfOne(int n) {
        // 记录数字中1的位数
        int result = 0;

        // JAVA语言规范中，int整形占四个字节，总计32位
        // 对每一个位置与1进行求与操作，再累加就可以求出当前数字的表示是多
```

```
少位1
        for (int i = 0; i < 32; i++) {
            result += (n & 1);
            n >>>= 1;
        }

        // 返回求得的结果
        return result;
    }

    /**
     * @param n 待的数字
     * @return 数字中二进制表表的1的数目
     */
    public static int numberOfOne2(int n) {
        // 记录数字中1的位数
        int result = 0;

        // 数字的二进制表示中有多少个1就进行多少次操作
        while (n != 0) {
            result++;
            // 从最右边的1开始，每一次操作都使n的最右的一个1变成了0，
            // 即使是符号位也会进行操作。
            n = (n - 1) & n;
        }

        // 返回求得的结果
        return result;
    }
}
```

一、题目

输入数字n，按顺序打印出从1到n位最大十进数的数值。比如输入3，则打印出1、2、3一直到最大三位数即999。

二、解题思路

①使用一个n位的数组来存储每一位的元素。例如n位3,则000表示为[0,0,0]。

使用递归的方式，存放每一位元素值。

②同上，使用一个n位的数组来存储每一位的元素。然后循环执行加1运算，并在数组中进行模拟进位，直到最高位需要进位，则表示循环结束。

三、解题代码

```
public class Test {

    /**
     * 输入数字n，按顺序打印出从1最大的n位十进制数。比如输入3，则打印出1、
2、3 一直到最大的3位数即999。
     *
     * @param n 数字的最大位数
     */
    public static void printOneToNthDigits(int n) {
        // 输入的数字不能为小于1
        if (n < 1) {
            throw new RuntimeException("The input number must la
rger than 0");
        }
        // 创建一个数组用于打存放值
        int[] arr = new int[n];
        printOneToNthDigits(0, arr);
    }

    /**
     * 输入数字n，按顺序打印出从1最大的n位十进制数。
```

```
 *
 * @param n   当前处理的是第个元素，从0开始计数
 * @param arr 存放结果的数组
 */
public static void printOneToNthDigits(int n, int[] arr) {

    // 说明数组已经装满元素
    if (n >= arr.length) {
        // 可以输出数组的值
        printArray(arr);
    } else {
        for (int i = 0; i <= 9; i++) {
            arr[n] = i;
            printOneToNthDigits(n + 1, arr);
        }
    }
}

/**
 * 输入数组的元素，从左到右，从第一个非0值到开始输出到最后的元素。
 *
 * @param arr 要输出的数组
 */
public static void printArray(int[] arr) {
    // 找第一个非0的元素
    int index = 0;
    while (index < arr.length && arr[index] == 0) {
        index++;
    }

    // 从第一个非0值到开始输出到最后的元素。
    for (int i = index; i < arr.length; i++) {
        System.out.print(arr[i]);
    }
    // 条件成立说明数组中有非零元素，所以需要换行
    if (index < arr.length) {
        System.out.println();
    }
}
```

```
    /**
     * 输入数字n，按顺序打印出从1最大的n位十进制数。比如输入3，则打印出1、
2、3 一直到最大的3位数即999。
     * 【第二种方法，比上一种少用内存空间】
     *
     * @param n 数字的最大位数
     */
    public static void printOneToNthDigits2(int n) {
        // 输入值必须大于0
        if (n < 1) {
            throw new RuntimeException("The input number must la
rger than 0");
        }

        // 创建一个长度为n的数组
        int[] arr = new int[n];
        // 为数组元素赋初始值
        for (int i = 0; i < arr.length; i++) {
            arr[i] = 0;
        }

        // 求结果，如果最高位没有进位就一直进行处理
        while (addOne(arr) == 0) {
            printArray(arr);
        }
    }

    /**
     * 对arr表示的数组的最低位加1 arr中的每个数都不能超过9不能小于0，每个
位置模拟一个数位
     *
     * @param arr 待加数组
     * @return 判断最高位是否有进位，如果有进位就返回1，否则返回0
     */
    public static int addOne(int[] arr) {
        // 保存进位值，因为每次最低位加1
        int carry = 1;
        // 最低位的位置的后一位
        int index = arr.length;
```

```
        do {
            // 指向上一个处理位置
            index--;
            // 处理位置的值加上进位的值
            arr[index] += carry;
            // 求处理位置的进位
            carry = arr[index] / 10;
            // 求处理位置的值
            arr[index] %= 10;
        } while (carry != 0 && index > 0);

        // 如果index=0说明已经处理了最高位，carry>0说明最高位有进位，返
回1
        if (carry > 0 && index == 0) {
            return 1;
        }

        // 无进位返回0
        return 0;
    }
}
```

一、题目

给定单向链表的一个头指针和节点指针，定义一个函数在O(1)时间删除该节点。

二、解题思路

由于给定了节点指针，那么要删除该节点。只要把该节点的值替换为下一个节点的值，同时让该节点直接指向下一个节点的下一个节点。相当于顶包代替了下一个节点，该节点自然就不存在。

需要注意的是如果指定节点是头结点，那么直接把头结点定义为下一个节点即可。如果是尾节点，需要循环遍历到该节点，然后让尾节点的上一个节点的指针为空即可。

三、解题代码

```
public class Test {
    /**
     * 链表结点
     */
    public static class ListNode {
        int value; // 保存链表的值
        ListNode next; // 下一个结点
    }

    /**
     * 给定单向链表的头指针和一个结点指针，定义一个函数在O(1)时间删除该结点
,
     * 【注意1：这个方法和文本上的不一样，书上的没有返回值，这个因为JAVA引
用传递的原因，
     * 如果删除的结点是头结点，如果不采用返回值的方式，那么头结点永远删除不
了】
     * 【注意2：输入的待删除结点必须是待链表中的结点，否则会引起错误，这个
条件由用户进行保证】
     *
     * @param head         链表表的头
```

```
     * @param toBeDeleted 待删除的结点
     * @return 删除后的头结点
     */
    public static ListNode deleteNode(ListNode head, ListNode to
BeDeleted) {

        // 如果输入参数有空值就返回表头结点
        if (head == null || toBeDeleted == null) {
            return head;
        }

        // 如果删除的是头结点，直接返回头结点的下一个结点
        if (head == toBeDeleted) {
            return head.next;
        }

        // 下面的情况链表至少有两个结点

        // 在多个节点的情况下，如果删除的是最后一个元素
        if (toBeDeleted.next == null) {
            // 找待删除元素的前驱
            ListNode tmp = head;
            while (tmp.next != toBeDeleted) {
                tmp = tmp.next;
            }
            // 删除待结点
            tmp.next = null;

        }
        // 在多个节点的情况下，如果删除的是某个中间结点
        else {
            // 将下一个结点的值输入当前待删除的结点
            toBeDeleted.value = toBeDeleted.next.value;
            // 待删除的结点的下一个指向原先待删除引号的下下个结点，即将待
删除的下一个结点删除
            toBeDeleted.next = toBeDeleted.next.next;
        }

        // 返回删除节点后的链表头结点
        return head;
```

```
    }
}
```

一、题目

输入一个整数数组，实现一个函数来调整该数组中数字的顺序，使得所有奇数位于数组的前半部分，所有偶数位予数组的后半部分。

二、解题思路

这个题目要求把奇数放在数组的前半部分， 偶数放在数组的后半部分，因此所有的奇数应该位于偶数的前面。也就是说我们在扫描这个数组的时候， 如果发现有偶数出现在奇数的前面，我们可以交换它们的顺序，交换之后就符合要求了。

因此我们可以维护两个指针，第一个指针初始化时指向数组的第一个数字，它只向后移动：第二个指针初始化时指向数组的最后一个数字， 它只向前移动。在两个指针相遇之前，第一个指针总是位于第二个指针的前面。如果第一个指针指向的数字是偶数，并且第二个指针指向的数字是奇数，我们就交换这两个数字。

三、解题代码

```
public class Test {

    /**
     * 输入一个整数数组，实现一个函数来调整该数组中数字的顺序，
     * 使得所有奇数位于数组的前半部分，所有偶数位予数组的后半部分。
     *
     * @param arr 输入的数组
     */
    public static void reorderOddEven(int[] arr) {
        // 对于输入的数组为空，或者长度小于2的只接返回
        if (arr == null || arr.length < 2) {
            return;
        }

        // 从左向右记录偶数的位置
        int start = 0;
        // 从右向左记录奇数的位置
        int end = arr.length - 1;
        // 开始调整奇数和偶数的位置
        while (start < end) {
            // 找偶数
            while (start < end && arr[start] % 2 != 0) {
                start++;
            }
            // 找奇数
            while (start < end && arr[end] % 2 == 0) {
                end--;
            }

            // 找到后就将奇数和偶数交换位置
            // 对于start=end的情况，交换不会产生什么影响
            // 所以将if判断省去了
            int tmp = arr[start];
            arr[start] = arr[end];
            arr[end] = tmp;
        }
    }
}
```

一、题目

输入一个链表，输出该链表中倒数第k 个结点。为了符合大多数人的习惯，本题从1 开始计数，即链表的尾结点是倒数第1 个结点．例如一个链表有6 个结点，从头结点开始它们的值依次是1 、2、3、4、5 、6。这个个链表的倒数第3 个结点是值为4 的结点。

二、解题思路

为了实现只遍历链表一次就能找到倒数第k 个结点，我们可以定义两个指针。第一个指针从链表的头指针开始遍历向前走k-1步，第二个指针保持不动；从第k 步开始，第二个指针也开始从链表的头指针开始遍历。由于两个指针的距离保持在k-1 ， 当第一个（走在前面的）指针到达链表的尾结点时，第二个指针（走在后面的）指针正好是倒数第k 个结点。

三、解题代码

```
public class Test {
    public static class ListNode {
        int value;
        ListNode next;
    }

    /**
     * 输入一个键表，输出该链表中倒数第k 个结点．为了符合大多数人的习惯，
     * 本题从1开始计数，即链表的尾结点是倒数第1个结点．例如一个链表有6个结
点，
     * 从头结点开始它们的值依次是1、2、3、4、5 6。这个链表的倒数第3个结点
是值为4的结点．
     *
     * @param head 链表的头结点
     * @param k     倒数第k个结点
     * @return 倒数第k个结点
     */
    public static ListNode findKthToTail(ListNode head, int k) {
```

```
        // 输入的链表不能为空，并且k大于0
        if (k < 1 || head == null) {
            return null;
        }

        // 指向头结点
        ListNode pointer = head;

        // 倒数第k个结点与倒数第一个结点相隔k-1个位置
        // pointer先走k-1个位置
        for (int i = 1; i < k; i++) {
            // 说明还有结点
            if (pointer.next != null) {
                pointer = pointer.next;
            }
            // 已经没有节点了，但是i还没有到达k-1说明k太大，链表中没有那
么多的元素
            else {
                // 返回结果
                return null;
            }

        }

        // pointer还没有走到链表的末尾，那么pointer和head一起走，
        // 当pointer走到最后一个结点即，pointer.next=null时，head就是
倒数第k个结点
        while (pointer.next != null) {
            head = head.next;
            pointer = pointer.next;
        }

        // 返回结果
        return head;
    }
}
```

时间： 2018-05-12 01:44:07

一、题目

定义一个函数，输入一个链表的头结点，反转该链表并输出反转后链表的头结点。

二、解题思路

①遍历。将指向下一个节点的指针指向上一个节点。

②递归。先让指向下一个节点的指针为空，然后递归调用，最后再将指向下一个节点的指针指向上一个节点。

三、解题代码

遍历

```
/**
 * 反转单链表
 * @param head
 * @return
 */
private static Node reverseHead(Node head) {
    if (head == null) {
        return head;
    }

    Node pre = head;
    Node cur = head.nextNode;
    Node next = null;
    while(cur != null){
        next = cur.nextNode;
        cur.nextNode = pre;

        pre = cur;
        cur = next;
    }
    head.nextNode = null;
    head = pre;
    return head;
}
```

递归

```
    /**
     * 递归反转
     * @param head
     * @return
     */
    private static Node reverseByRecur(Node current) {
        if (current == null || current.nextNode == null) return
current;
        Node nextNode = current.nextNode;
        current.nextNode = null;
        Node reverseRest = reverseByRecur(nextNode);
        nextNode.nextNode = current;
        return reverseRest;
    }
```

一、题目

输入两个递增排序的链表，合并这两个链表并使新链表中的结点仍然是按照递增排序的。

二、解题思路

Step1.定义一个指向新链表的指针，暂且让它指向NULL；

Step2.比较两个链表的头结点，让较小的头结点作为新链表的头结点；

Step3.有两种方法。

①递归比较两个链表的其余节点，让较小的节点作为上一个新节点的后一个节点；

②循环比较两个链表的其余节点，让较小的节点作为上一个新节点的后一个节点。直到有一个链表没有节点，然后将新链表的最后一个节点直接指向剩余链表的节点。

三、解题代码

```
public class Test {
    public static class ListNode {
        int value;
        ListNode next;
    }

    /**
     * 输入两个递增排序的链表，合并这两个链表并使新链表中的结点仍然是按照递
增排序的
     *
     * @param head1 第一个有序链表
     * @param head2 第二个有序链表
     * @return 合并后的有序链表头
     */
    public static ListNode merge(ListNode head1, ListNode head2)
{
```

```
        // 如果第一个链表为空，返回第二个链表头结点
        if (head1 == null) {
            return head2;
        }

        // 如果第二个结点为空，返回第一个链表头结点
        if (head2 == null) {
            return head1;
        }

        // 创建一个临时结点，用于添加元素时方便
        ListNode root = new ListNode();
        // 用于指向合并后的新链的尾结点
        ListNode pointer = root;

        // 当两个链表都不为空就进行合并操作
        while (head1 != null && head2 != null) {
            // 下面的操作合并较小的元素
            if (head1.value < head2.value) {
                pointer.next = head1;
                head1 = head1.next;
            } else {
                pointer.next = head2;
                head2 = head2.next;
            }

            // 将指针移动到合并后的链表的末尾
            pointer = pointer.next;
        }

        // 下面的两个if有且只一个if会内的内容会执行

        // 如果第一个链表的元素未处理完将其，接到合并链表的最后一个结点之后
        if (head1 != null) {
            pointer.next = head1;
        }

        // 如果第二个链表的元素未处理完将其，接到合并链表的最后一个结点之后
        if (head2 != null) {
            pointer.next = head2;
```

```
        }

        // 返回处理结果
        return root.next;
    }


    /**
     * 输入两个递增排序的链表，合并这两个链表并使新链表中的结点仍然是按照递
增排序的
     * 【使用的是递归的解法，不推荐，递归调用的时候会有方法入栈，需要更多的
内存】
     *
     * @param head1 第一个有序链表
     * @param head2 第二个有序链表
     * @return 合并后的有序链表头
     */
    public static ListNode merge2(ListNode head1, ListNode head2)
 {
        // 如果第一个链表为空，返回第二个链表头结点
        if (head1 == null) {
            return head2;
        }

        // 如果第二个链表为空，返回第一个链表头结点
        if (head2 == null) {
            return head1;
        }

        // 记录两个链表中头部较小的结点
        ListNode tmp = head1;
        if (tmp.value < head2.value) {
            // 如果第一个链表的头结点小，就递归处理第一个链表的下一个结点
和第二个链表的头结点
            tmp.next = merge2(head1.next, head2);
        } else {
            // 如果第二个链表的头结点小，就递归处理第一个链表的头结点和第
二个链表的头结点的下一个结点
            tmp = head2;
            tmp.next = merge2(head1, head2.next);
```

```
            }

            // 返回处理结果
            return tmp;
        }
    }
```

一、题目

输入两棵二叉树A 和B，判断B 是不是A 的子结构。

二、解题思路

要查找树A 中是否存在和树B 结构一样的子树，我们可以分成两步： 第一步在树A 中找到和B 的根结点的值一样的结点R， 第二步再判断树A 中以R 为根结点的子树是不是包含和树B 一样的结构。

三、解题代码

```
public class Test {
    /**
     * 二叉树的树结点
     */
    public static class BinaryTreeNode {
        int value;
        BinaryTreeNode left;
        BinaryTreeNode right;
    }

    /**
     * 输入两棵二叉树A和B，判断B是不是A的子结构。
     * 该方法是在A树中找到一个与B树的根节点相等的元素的结点，
     * 从这个相等的结点开始判断树B是不是树A的子结构，如果找到其的一个就返回
，
     * 否则直到所有的结点都找完为止。
     *
     * @param root1 树A的根结点
     * @param root2 树B的根结点
     * @return true：树B是树A的子结构，false：树B是不树A的子结构
     */
    public static boolean hasSubtree(BinaryTreeNode root1, Binar
yTreeNode root2) {
        // 只要两个对象是同一个就返回true
```

```
        if (root1 == root2) {
            return true;
        }

        // 只要树B的根结点点为空就返回true
        if (root2 == null) {
            return true;
        }

        // 树B的根结点不为空，如果树A的根结点为空就返回false
        if (root1 == null) {
            return false;
        }

        // 记录匹配结果
        boolean result = false;

        // 如果结点的值相等就，调用匹配方法
        if (root1.value == root2.value) {
            result = match(root1, root2);
        }

        // 如果匹配就直接返回结果
        if (result) {
            return true;
        }

        // 如果不匹配就找树A的左子结点和右子结点进行判断
        return hasSubtree(root1.left, root2) || hasSubtree(root1
.right, root2);
    }

    /**
     * 从树A根结点root1和树B根结点root2开始，一个一个元素进行判断，判断B
是不是A的子结构
     *
     * @param root1 树A开始匹配的根结点
     * @param root2 树B开始匹配的根结点
     * @return 树B是树A的子结构，false：树B是不树A的子结构
     */
```

```
    public static boolean match(BinaryTreeNode root1, BinaryTree
Node root2) {
        // 只要两个对象是同一个就返回true
        if (root1 == root2) {
            return true;
        }

        // 只要树B的根结点点为空就返回true
        if (root2 == null) {
            return true;
        }
        // 树B的根结点不为空，如果树A的根结点为空就返回false
        if (root1 == null) {
            return false;
        }

        // 如果两个结点的值相等，则分别判断其左子结点和右子结点
        if (root1.value == root2.value) {
            return match(root1.left, root2.left) && match(root1.
right, root2.right);
        }

        // 结点值不相等返回false
        return false;
    }

}
```

一、题目

请完成一个函数，输入一个二叉树，该函数输出它的镜像。

二、解题思路

先前序遍历这棵树的每个结点，如果遍历到的结点有子结点，就交换它的两个子结点。当交换完所有非叶子结点的左右子结点之后，就得到了树的镜像。

三、解题代码

```
public class Test {
    /**
     * 二叉树的树结点
     */
    public static class BinaryTreeNode {
        int value;
        BinaryTreeNode left;
        BinaryTreeNode right;
    }

    /**
     * 请完成一个函数，输入...个二叉树，该函数输出它的镜像
     *
     * @param node 二叉树的根结点
     */
    public static void mirror(BinaryTreeNode node) {
        // 如果当前结点不为空则进行操作
        if (node != null) {
            // 下面是交换结点左右两个子树
            BinaryTreeNode tmp = node.left;
            node.left = node.right;
            node.right = tmp;

            // 对结点的左右两个子树进行处理
            mirror(node.left);
            mirror(node.right);
        }
    }
}
```

一、题目

输入一个矩阵，按照从外向里以顺时针的顺序依次扫印出每一个数字。

二、解题思路

把打印一圈分为四步：第一步从左到右打印一行，第二步从上到下打印一列，第三步从右到左打印一行，第四步从下到上打印一列。每一步我们根据起始坐标和终止坐标用一个循环就能打印出一行或者一列。

不过值得注意的是，最后一圈有可能退化成只有一行、只有一列，甚至只有一个数字，因此打印这样的一圈就不再需要四步。

因此我们要仔细分析打印时每一步的前提条件。第一步总是需要的， 因为打印一圈至少有一步。如果只有一行，那么就不用第二步了。也就是需要第二步的前提条件是终止行号大于起始行号。需要第三步打印的前提条件是圈内至少有两行两列，也就是说除了要求终止行号大于起始行号之外，还要求终止列号大于起始列号。同理，需要打印第四步的前提条件是至少有三行两列，因此要求终止行号比起始行号至少大2 ， 同时终止列号大于起始列号。

三、解题代码

```
public class Test {
    /**
     * 输入一个矩阵，按照从外向里以顺时针的顺序依次打印每一个数字
     *
     * @param numbers 输入的二维数组，二维数组必须是N*M的，否则分出错
     */
    public static void printMatrixClockWisely(int[][] numbers) {
        // 输入的参数不能为空
        if (numbers == null) {
            return;
        }

        // 记录一圈（环）的开始位置的行
        int x = 0;
```

```
        // 记录一圈（环）的开始位置的列
        int y = 0;
        // 对每一圈（环）进行处理，
        // 行号最大是(numbers.length-1)/2
        // 列号最大是(numbers[0].length-1)/2
        while (x * 2 < numbers.length && y * 2 < numbers[0].leng
th) {
            printMatrixInCircle(numbers, x, y);
            // 指向下一个要处理的的环的第一个位置
            x++;
            y++;
        }
    }

    public static void printMatrixInCircle(int[][] numbers, int
x, int y) {
        // 数组的行数
        int rows = numbers.length;
        // 数组的列数
        int cols = numbers[0].length;

        // 输出环的上面一行，包括最中的那个数字
        for (int i = y; i <= cols - y - 1; i++) {
            System.out.print(numbers[x][i] + " ");
        }

        // 环的高度至少为2才会输出右边的一列
        // rows-x-1：表示的是环最下的那一行的行号
        if (rows - x - 1 > x) {
            // 因为右边那一列的最上面那一个已经被输出了，所以行呈从x+1开
始，
            // 输出包括右边那列的最下面那个
            for (int i = x + 1; i <= rows - x - 1; i++) {
                System.out.print(numbers[i][cols - y - 1] + " ")
;
            }
        }

        // 环的高度至少是2并且环的宽度至少是2才会输出下面那一行
        // cols-1-y：表示的是环最右那一列的列号
```

```
        if (rows - x - 1 > x && cols - 1 - y > y) {
            // 因为环的左下角的位置已经输出了，所以列号从cols-y-2开始
            for (int i = cols - y - 2; i >= y; i--) {
                System.out.print(numbers[rows - 1 - x][i] + " ")
;
            }
        }

        // 环的宽度至少是2并且环的高度至少是3才会输出最左边那一列
        // rows-x-1：表示的是环最下的那一行的行号
        if (cols - 1 - y > y && rows - 1 - x > x + 1) {
            // 因为最左边那一列的第一个和最后一个已经被输出了
            for (int i = rows - 1 - x - 1; i >= x + 1; i--) {
                System.out.print(numbers[i][y] + " ");
            }
        }
    }
}
```

一、题目

定义栈的数据结构，请在该类型中实现一个能够得到栈的最小数的min 函数。在该栈中，调用min、push 及pop的时间复杂度都是O(1)。

二、解题思路

把每次的最小元素（之前的最小元素和新压入战的元素两者的较小值）都保存起来放到另外一个辅助栈里。

如果每次都把最小元素压入辅助栈， 那么就能保证辅助栈的栈顶一直都是最小元素．当最小元素从数据栈内被弹出之后，同时弹出辅助栈的栈顶元素，此时辅助栈的新栈顶元素就是下一个最小值。

三、解题代码

```
public class MinStack {

    private Stack<Integer> stack = new Stack<Integer>();
    private Stack<Integer> minStack = new Stack<Integer>(); //辅
助栈：栈顶永远保存stack中当前的最小的元素


    public void push(int data) {
        stack.push(data);  //直接往栈中添加数据

        //在辅助栈中需要做判断
        if (minStack.size() == 0 || data < minStack.peek()) {
            minStack.push(data);
        } else {
            minStack.add(minStack.peek());   //【核心代码】peek方法
返回的是栈顶的元素
        }
    }

    public int pop() throws Exception {
        if (stack.size() == 0) {
            throw new Exception("栈中为空");
        }

        int data = stack.pop();
        minStack.pop();  //核心代码
        return data;
    }

    public int min() throws Exception {
        if (minStack.size() == 0) {
            throw new Exception("栈中空了");
        }
        return minStack.peek();
    }
}
```

一、题目

输入两个整数序列，第一个序列表示栈的压入顺序，请判断二个序列是否为该栈的弹出顺序。假设压入栈的所有数字均不相等。

二、解题思路

解决这个问题很直观的想法就是建立一个辅助栈，把输入的第一个序列中的数字依次压入该辅助栈，并按照第二个序列的顺序依次从该栈中弹出数字。

判断一个序列是不是栈的弹出序列的规律：如果下一个弹出的数字刚好是栈顶数字，那么直接弹出。如果下一个弹出的数字不在栈顶，我们把压栈序列中还没有入栈的数字压入辅助栈，直到把下一个需要弹出的数字压入栈顶为止。如果所有的数字都压入栈了仍然没有找到下一个弹出的数字，那么该序列不可能是一个弹出序列。

三、解题代码

```
public class StackTest {


    //方法：data1数组的顺序表示入栈的顺序。现在判断data2的这种出栈顺序是否正确
    public static boolean sequenseIsPop(int[] data1, int[] data2) {
        Stack<Integer> stack = new Stack<Integer>(); //这里需要用到辅助栈

        for (int i = 0, j = 0; i < data1.length; i++) {
            stack.push(data1[i]);

            while (stack.size() > 0 && stack.peek() == data2[j]) {
                stack.pop();
                j++;
            }
        }
        return stack.size() == 0;
    }
}
```

一、题目

从上往下打印出二叉树的每个结点，同一层的结点按照从左向右的顺序打印。

二、解题思路

这道题实质是考查树的遍历算法。从上到下打印二叉树的规律：每一次打印一个结点的时候，如果该结点有子结点， 则把该结点的子结点放到一个队列的末尾。接下来到队列的头部取出最早进入队列的结点，重复前面的打印操作，直至队列中所有的结点都被打印出来为止。

三、解题代码

```
public class Test {
    /**
     * 二叉树的树结点
     */
    public static class BinaryTreeNode {
        int value;
        BinaryTreeNode left;
        BinaryTreeNode right;
    }

    /**
     * 从上往下打印出二叉树的每个结点，向一层的结点按照从左往右的顺序打印。
     * 例如下的二叉树，
     *       8
     *     /   \
     *    6     10
     *   /  \   / \
     * 5   7  9  11
     * 则依次打印出8、6、10、5、3 、9、11.
     *
     * @param root 树的结点
     */
    public static void printFromToBottom(BinaryTreeNode root) {
```

```
        // 当结点非空时才进行操作
        if (root != null) {
            // 用于存放还未遍历的元素
            Queue<BinaryTreeNode> list = new LinkedList<>();
            // 将根结点入队
            list.add(root);
            // 用于记录当前处理的结点
            BinaryTreeNode curNode;

            // 队列非空则进行处理
            while (!list.isEmpty()) {
                // 删除队首元素
                curNode = list.remove();
                // 输出队首元素的值
                System.out.print(curNode.value + " ");
                // 如果左子结点不为空，则左子结点入队
                if (curNode.left != null) {
                    list.add(curNode.left);
                }
                // 如果右子结点不为空，则左子结点入队
                if (curNode.right != null) {
                    list.add(curNode.right);
                }
            }
        }
    }
}
```

一、题目

输入一个整数数组，判断该数组是不是某二叉搜索树的后序遍历的结果。如果是则返回true。否则返回false。假设输入的数组的任意两个数字都互不相同。

二、解题思路

在后序遍历得到的序列中， 最后一个数字是树的根结点的值。数组中前面的数字可以分为两部分： 第一部分是左子树结点的值，它们都比根结点的值小： 第二部分是右子树结点的值，它们都比根结点的值大。

三、解题代码

```
public class Test {
    /**
     * 输入一个整数数组，判断该数组是不是某二叉搜索树的后序遍历的结果。
     * 如果是则返回true。否则返回false。假设输入的数组的任意两个数字都互不
相同。
     *
     * @param sequence 某二叉搜索树的后序遍历的结果
     * @return true：该数组是某二叉搜索树的后序遍历的结果。false：不是
     */
    public static boolean verifySequenceOfBST(int[] sequence) {

        // 输入的数组不能为空，并且有数据
        if (sequence == null || sequence.length <= 0) {
            return false;
        }

        // 有数据，就调用辅助方法
        return verifySequenceOfBST(sequence, 0, sequence.length
- 1);
    }

    /**
     * 输入一个整数数组，判断该数组是不是某二叉搜索树的后序遍历的结果。
```

```
     * @param sequence 某二叉搜索树的后序遍历的结果
     * @param start    处理的开始位置
     * @param end      处理的结束位置
     * @return true：该数组是某二叉搜索树的后序遍历的结果。false：不是
     */
    public static boolean verifySequenceOfBST(int[] sequence, int
 start, int end) {

        // 如果对应要处理的数据只有一个或者已经没有数据要处理（start>end
）就返回true
        if (start >= end) {
            return true;
        }

        // 从左向右找第一个不小于根结点（sequence[end]）的元素的位置
        int index = start;
        while (index < end - 1 && sequence[index] < sequence[end
]) {
            index++;
        }

        // 执行到此处[start, index-1]的元素都是小于根结点的（sequence[
end]）
        // [start, index-1]可以看作是根结点的左子树

        // right用于记录第一个大于根结点的元素的位置

        int right = index;

        // 接下来要保证[index, end-1]的所有元素都是大于根根点的值
        // 因为[index, end-1]是根结点的右子树
        // 从第一个不小于根结点的元素开始，找第一个不大于根结点的元素
        while (index < end - 1 && sequence[index] > sequence[end
]) {
            index++;
        }
        // 如果[index, end-1]中有小于等于根结点的元素，
        // 不符合二叉搜索树的定义，返回false
        if (index != end - 1) {
            return false;
```

```
        }

        // 执行到此处说明直到目前为止，还是合法的
        // [start, index-1]为根结点左子树的位置
        // [index, end-1]为根结点右子树的位置
        index = right;
        return verifySequenceOfBST(sequence, start, index - 1) &
& verifySequenceOfBST(sequence, index, end - 1);
    }
}
```

一、题目

输入一棵二叉树和一个整数， 打印出二叉树中结点值的和为输入整数的所有路径。从树的根结点开始往下一直到叶结点所经过的结点形成一条路径。

二、解题思路

由于路径是从根结点出发到叶结点， 也就是说路径总是以根结点为起始点，因此我们首先需要遍历根结点。在树的前序、中序、后序三种遍历方式中，只有前序遍历是首先访问根结点的。

当用前序遍历的方式访问到某一结点时， 我们把该结点添加到路径上，并累加该结点的值。如果该结点为叶结点并且路径中结点值的和刚好等于输入的整数， 则当前的路径符合要求，我们把它打印出来。如果当前结点不是叶结点，则继续访问它的子结点。当前结点访问结束后，递归函数将自动回到它的父结点。因此我们在函数退出之前要在路径上删除当前结点并减去当前结点的值，以确保返回父结点时路径刚好是从根结点到父结点的路径。

不难看出保存路径的数据结构实际上是一个栈， 因为路径要与递归调用状态一致，而递归调用的本质就是一个压栈和出栈的过程。

三、解题代码

```
public class Test {
    /**
     * 二叉树的树结点
     */
    public static class BinaryTreeNode {
        int value;
        BinaryTreeNode left;
        BinaryTreeNode right;
    }

    /**
     * 输入一棵二叉树和一个整数， 打印出二叉树中结点值的和为输入整数的所有
路径。
```

```
     * 从树的根结点开始往下一直到叶结点所经过的结点形成一条路径。
     *
     * @param root        树的根结点
     * @param expectedSum 要求的路径和
     */
    public static void findPath(BinaryTreeNode root, int expecte
dSum) {
        // 创建一个链表，用于存放根结点到当前处理结点的所经过的结点
        List<Integer> list = new ArrayList<>();

        // 如果根结点不为空，就调用辅助处理方法
        if (root != null) {
            findPath(root, 0, expectedSum, list);
        }
    }

    /**
     * @param root        当前要处理的结点
     * @param curSum      当前记录的和（还未加上当前结点的值）
     * @param expectedSum 要求的路径和
     * @param result      根结点到当前处理结点的所经过的结点，（还未包括
当前结点）
     */
    public static void findPath(BinaryTreeNode root, int curSum,
int expectedSum, List<Integer> result) {

        // 如果结点不为空就进行处理
        if (root != null) {
            // 加上当前结点的值
            curSum += root.value;
            // 将当前结点入队
            result.add(root.value);
            // 如果当前结点的值小于期望的和
            if (curSum < expectedSum) {
                // 递归处理左子树
                findPath(root.left, curSum, expectedSum, result)
;
                // 递归处理右子树
                findPath(root.right, curSum, expectedSum, result
);
```

```
            }
            // 如果当前和与期望的和相等
            else if (curSum == expectedSum) {
                // 当前结点是叶结点，则输出结果
                if (root.left == null && root.right == null) {
                    System.out.println(result);
                }
            }
            // 移除当前结点
            result.remove(result.size() - 1);
        }
    }
}
```

一、题目

请实现函数ComplexListNode clone(ComplexListNode head),复制一个复杂链表。在复杂链表中，每个结点除了有一个next 域指向下一个结点外，还有一个sibling 指向链表中的任意结点或者null。

二、解题思路

在不用辅助空间的情况下实现O(n)的时间效率。

第一步：仍然是根据原始链表的每个结点N 创建对应的N'。把N'链接在N的后面。

第二步：设置复制出来的结点的sibling。假设原始链表上的N的sibling指向结点S，那么其对应复制出来的N'是N的next指向的结点，同样S'也是S的next指向的结点。

第三步：把这个长链表拆分成两个链表。把奇数位置的结点用next . 链接起来就是原始链表，把偶数位置的结点用next 链接起来就是复制 出来的链表。

三、解题代码

```
public class Test {
    /**
     * 复杂链表结点
     */
    public static class ComplexListNode {
        int value;
        ComplexListNode next;
        ComplexListNode sibling;
    }

    /**
     * 实现函复制一个复杂链表。在复杂链表中，每个结点除了有一个next字段指向
下一个结点外，
     * 还有一个sibling字段指向链表中的任意结点或者NULL
     *
     * @param head 链表表头结点
     * @return 复制结点的头结点
```

```
     */
    public static ComplexListNode clone(ComplexListNode head) {
        // 如果链表为空就直接返回空
        if (head == null) {
            return null;
        }

        // 先复制结点
        cloneNodes(head);
        // 再链接sibling字段
        connectNodes(head);
        // 将整个链表拆分，返回复制链表的头结点
        return reconnectNodes(head);
    }

    /**
     * 复制一个链表，并且将复制后的结点插入到被复制的结点后面，只链接复制结
点的next字段
     *
     * @param head 待复制链表的头结点
     */
    public static void cloneNodes(ComplexListNode head) {
        // 如果链表不空，进行复制操作
        while (head != null) {
            // 创建一个新的结点
            ComplexListNode tmp = new ComplexListNode();
            // 将被复制结点的值传给复制结点
            tmp.value = head.value;
            // 复制结点的next指向下一个要被复制的结点
            tmp.next = head.next;
            // 被复制结点的next指向复制结点
            head.next = tmp;
            // 到些处就已经完成了一个结点的复制并且插入到被复制结点的后面
            // heed指向下一个被复制结点的位置
            head = tmp.next;
        }
    }

    /**
     * 设置复制结点的sibling字段
```

```
     *
     * @param head 链表的头结
     */
    public static void connectNodes(ComplexListNode head) {
        // 如链表不为空
        while (head != null) {
            // 当前处理的结点sibling字段不为空，则要设置其复制结点的sibl
ing字段
            if (head.sibling != null) {
                // 复制结点的sibling指向被复制结点的sibling字段的下一
个结点
                // head.next：表求复制结点，
                // head.sibling：表示被复制结点的sibling所指向的结点，

                // 它的下一个结点就是它的复制结点
                head.next.sibling = head.sibling.next;
            }
            // 指向下一个要处理的复制结点
            head = head.next.next;
        }
    }

    /**
     * 刚复制结点和被复制结点拆开，还原被复制的链表，同时生成复制链表
     *
     * @param head 链表的头结点
     * @return 复制链表的头结点
     */
    public static ComplexListNode reconnectNodes(ComplexListNode
 head) {

        // 当链表为空就直接返回空
        if (head == null) {
            return null;
        }

        // 用于记录复制链表的头结点
        ComplexListNode newHead = head.next;
        // 用于记录当前处理的复制结点
        ComplexListNode pointer = newHead;
```

```
        // 被复制结点的next指向下一个原链表结点
        head.next = newHead.next;
        // 指向新的被复制结点
        head = head.next;

        while (head != null) {
            // pointer指向复制结点
            pointer.next = head.next;
            pointer = pointer.next;
            // head的下一个指向复制结点的下一个结点，即原来链表的结点
            head.next = pointer.next;
            // head指向下一个原来链表上的结点
            head = pointer.next;
        }

        // 返回复制链表的头结点
        return newHead;
    }
}
```

一、题目

输入一棵二叉搜索树，将该二叉搜索树转换成一个排序的双向链表。要求不能创建任何新的结点，只能调整树中结点指针的指向。

二、解题思路

在二叉树中，每个结点都有两个指向子结点的指针。在双向链表中，每个结点也有两个指针，它们分别指向前一个结点和后一个结点。由于这两种结点的结构相似，同时二叉搜索树也是一种排序的数据结构，因此在理论上有可能实现二叉搜索树和排序的双向链表的转换。

在搜索二叉树中，左子结点的值总是小于父结点的值，右子结点的值总是大于父结点的值。因此我们在转换成排序双向链表时，原先指向左子结点的指针调整为链表中指向前一个结点的指针，原先指向右子结点的指针调整为链表中指向后一个结点指针。接下来我们考虑该如何转换。

由于要求转换之后的链表是排好序的，我们可以中序遍历树中的每一个结点， 这是因为中序遍历算法的特点是按照从小到大的顺序遍历二叉树的每一个结点。当遍历到根结点的时候，我们把树看成三部分：根结点，左子树，右子树。根据排序链表的定义，根结点将和它的左子树的最大一个结点链接起来，同时它还将和右子树最小的结点链接起来。

三、解题代码

```
public class Test {
    /**
     * 二叉树的树结点
     */
    public static class BinaryTreeNode {
        int value;
        BinaryTreeNode left;
        BinaryTreeNode right;
    }
```

```
    /**
     * 题目：输入一棵二叉搜索树，将该二叉搜索树转换成一个排序的双向链表。
     * 要求不能创建任何新的结点，只能调整树中结点指针的指向。
     *
     * @param root 二叉树的根结点
     * @return 双向链表的头结点
     */
    public static BinaryTreeNode convert(BinaryTreeNode root) {

        // 用于保存处理过程中的双向链表的尾结点
        BinaryTreeNode[] lastNode = new BinaryTreeNode[1];
        convertNode(root, lastNode);

        // 找到双向链表的头结点
        BinaryTreeNode head = lastNode[0];
        while (head != null && head.left != null) {
            head = head.left;
        }
        return head;
    }


    /**
     * 链表表转换操作
     *
     * @param node      当前的根结点
     * @param lastNode 已经处理好的双向链表的尾结点，使用一个长度为1的数
组，类似C++中的二级指针
     */
    public static void convertNode(BinaryTreeNode node, BinaryTr
eeNode[] lastNode) {
        // 结点不为空
        if (node != null) {

            // 如果有左子树就先处理左子树
            if (node.left != null) {
                convertNode(node.left, lastNode);
            }

            // 将当前结点的前驱指向已经处理好的双向链表（由当前结点的左子
```

```
树构成）的尾结点
            node.left = lastNode[0];

            // 如果左子树转换成的双向链表不为空，设置尾结点的后继
            if (lastNode[0] != null) {
                lastNode[0].right = node;
            }

            // 记录当前结点为尾结点
            lastNode[0] = node;

            // 处理右子树
            if (node.right != null) {
                convertNode(node.right, lastNode);
            }
        }
    }
}
```

一、题目

输入一个字符串，打印出该字符串中字符的所有排列。例如输入字符串abc。则打印出由字符a、b、c 所能排列出来的所有字符串abc、acb、bac 、bca、cab 和cba 。

二、解题思路

把一个字符串看成由两部分组成：第一部分为它的第一个字符，第二部分是后面的所有字符。

我们求整个字符串的排列，可以看成两步：首先求所有可能出现在第一个位置的字符，即把第一个字符和后面所有的字符交换。这个时候我们仍把后面的所有字符分成两部分：后面字符的第一个字符，以及这个字符之后的所有字符。

这其实是很典型的递归思路。

三、解题代码

```
public class Test {
    /**
     * 题目：输入一个字符串，打印出该字符事中字符的所有排列。例如输入字符串
abc。
     * 则打印出由字符a、b、c 所能排列出来的所有字符串abc、acb、bac、bca
、cab和cba。
     *
     * @param chars 待排序的字符数组
     */
    public static void permutation(char[] chars) {
        // 输入较验
        if (chars == null || chars.length < 1) {
            return;
        }
        // 进行排列操作
        permutation(chars, 0);
    }
```

```
    /**
     * 求字符数组的排列
     *
     * @param chars 待排列的字符串
     * @param begin 当前处理的位置
     */
    public static void permutation(char[] chars, int begin) {
        // 如果是最后一个元素了，就输出排列结果
        if (chars.length - 1 == begin) {
            System.out.print(new String(chars) + " ");
        } else {
            char tmp;
            // 对当前还未处理的字符串进行处理，每个字符都可以作为当前处理
位置的元素
            for (int i = begin; i < chars.length; i++) {
                // 下面是交换元素的位置
                tmp = chars[begin];
                chars[begin] = chars[i];
                chars[i] = tmp;

                // 处理下一个位置
                permutation(chars, begin + 1);
            }
        }
    }
}
```

一、题目

数组中有一个数字出现的次数超过数组长度的一半，请找出这个数字。

二、解题思路

解法一：基于**Partition** 函数的**O(n)**算法

数组中有一个数字出现的次数超过了数组长度的一半。如果把这个数组排序，那么排序之后位于数组中间的数字一定就是那个出现次数超过数组长度一半的数字。也就是说，这个数字就是统计学上的中位数，即长度为n 的数组中第n/2 大的数字。

这种算法是受快速排序算法的启发。在随机快速排序算法中，我们先在数组中随机选择一个数字，然后调整数组中数字的顺序， 使得比选中的数字小数字都排在它的左边，比选中的数字大的数字都排在它的右边。如果这个选中的数字的下标刚好是n/2，那么这个数字就是数组的中位数。如果它的下标大于n/2 ，那么中位数应该位于它的左边，我们可以接着在它的左边部分的数组中查找。如果它的下标小于n/2，那么中位数应该位于它的右边，我们可以接着在它的右边部分的数组中查找。这是一个典型的递归过程。

解法二：根据数组组特点找出**O(n)**的算法

数组中有一个数字出现的次数超过数组长度的一半，也就是说它出现的次数比其他所有数字出现次数的和还要多。因此我们可以考虑在遍历数组的时候保存两个值：一个是数组中的一个数字， 一个是次数。当我们遍历到下一个数字的时候，如果下一个数字和我们之前保存的数字相同，则次数加1，如果下一个数字和我们之前保存的数字不同，则次数减1 。如果次数为零，我们需要保存下一个数字，并把次数设为1 。由于我们要找的数字出现的次数比其他所有数字出现的次数之和还要多，那么要找的数字肯定是最后一次把次数设为1 时对应的数字。

本题采用第二种实现方式

三、解题代码

```
public class Test {
```

```
    /**
     * 题目：数组中有一个数字出现的次数超过数组长度的一半，请找出这个数字
     *
     * @param numbers 输入数组
     * @return 找到的数字
     */
    public static int moreThanHalfNum(int[] numbers) {

        // 输入校验
        if (numbers == null || numbers.length < 1) {
            throw new IllegalArgumentException("array length mus
t large than 0");
        }

        // 用于记录出现次数大于数组一半的数
        int result = numbers[0];
        // 于当前记录的数不同的数的个数
        int count = 1;
        // 从第二个数开始向后找
        for (int i = 1; i < numbers.length; i++) {
            // 如果记数为0
            if (count == 0) {
                // 重新记录一个数，假设它是出现次数大于数组一半的
                result = numbers[i];
                // 记录统计值
                count = 1;
            }
            // 如果记录的值与统计值相等，记数值增加
            else if (result == numbers[i]) {
                count++;
            }
            // 如果不相同就减少，相互抵消
            else {
                count--;
            }
        }

        // 最后的result可能是出现次数大于数组一半长度的值
        // 统计result的出现次数
        count = 0;
```

```
        for (int number : numbers) {
            if (result == number) {
                count++;
            }
        }

        // 如果出现次数大于数组的一半就返回对应的值
        if (count > numbers.length / 2) {
            return result;
        }
        // 否则输入异常
        else {
            throw new IllegalArgumentException("invalid input");
        }
    }
}
```

一、题目

输入n个整数，找出其中最小的k个数。

例子说明：

例如输入4 、5 、1、6、2、7、3 、8 这8 个数字，则最小的4 个数字是1 、2、3 、4

二、解题思路

解法一：**O(n)**时间算法，只有可以修改输入数组时可用。

可以基于Partition函数来解决这个问题。如果基于数组的第k个数字来调整，使得比第k个数字小的所有数字都位于数组的左边，比第k个数字大的所有数字都位于数组的右边。这样调整之后，位于数组中左边的k个数字就是最小的k 个数字（这k 个数字不一定是排序的〉。

解法二： **O（nlogk）**的算法，精剧适合处理海量数据。

先创建一个大小为k的数据容器来存储最小的k个数字，接下来我们每次从输入的n个整数中读入一个数·如果容器中已有的数字少于k个，则直接把这次读入的整数放入容器之中：如果容器中己有k 数字了，也就是容器己满，此时我们不能再插入新的数字而只能替换已有的数字。找出这己有的k 个数中的最大值，然后1在这次待插入的整数和最大值进行比较。如果待插入的值比当前已有的最大值小，则用这个数替换当前已有的最大值：如果待插入的值比当前已有的最大值还要大，那么这个数不可能是最小的k个整数之一，于是我们可以抛弃这个整数。

因此当容器满了之后，我们要做3 件事情： 一是在k 个整数中找到最大数： 二是有可能在这个容器中删除最大数： 三是有可能要插入一个新的数字。我们可以使用一个大顶堆在**O(logk**）时间内实现这三步操作。

三、解题代码

```
public class Test {
    /**
     * 大顶堆
```

```
     *
     * @param <T> 参数化类型
     */
    private final static class MaxHeap<T extends Comparable<T>>
{
        // 堆中元素存放的集合
        private List<T> items;
        // 用于计数
        private int cursor;

        /**
         * 构造一个椎，始大小是32
         */
        public MaxHeap() {
            this(32);
        }

        /**
         * 造诣一个指定初始大小的堆
         *
         * @param size 初始大小
         */
        public MaxHeap(int size) {
            items = new ArrayList<>(size);
            cursor = -1;
        }

        /**
         * 向上调整堆
         *
         * @param index 被上移元素的起始位置
         */
        public void siftUp(int index) {
            T intent = items.get(index); // 获取开始调整的元素对象

            while (index > 0) { // 如果不是根元素
                int parentIndex = (index - 1) / 2; // 找父元素对象
的位置
                T parent = items.get(parentIndex);  // 获取父元素
对象
```

```
                if (intent.compareTo(parent) > 0) { //上移的条件，
子节点比父节点大
                    items.set(index, parent); // 将父节点向下放
                    index = parentIndex; // 记录父节点下放的位置
                } else { // 子节点不比父节点大，说明父子路径已经按从大
到小排好顺序了，不需要调整了
                    break;
                }
            }

            // index此时记录是的最后一个被下放的父节点的位置（也可能是自
身），所以将最开始的调整的元素值放入index位置即可
            items.set(index, intent);
        }

        /**
         * 向下调整堆
         *
         * @param index 被下移的元素的起始位置
         */
        public void siftDown(int index) {
            T intent = items.get(index);  // 获取开始调整的元素对象
            int leftIndex = 2 * index + 1; // // 获取开始调整的元素
对象的左子结点的元素位置

            while (leftIndex < items.size()) { // 如果有左子结点
                T maxChild = items.get(leftIndex); // 取左子结点的
元素对象，并且假定其为两个子结点中最大的
                int maxIndex = leftIndex; // 两个子节点中最大节点元
素的位置，假定开始时为左子结点的位置

                int rightIndex = leftIndex + 1;  // 获取右子结点的
位置
                if (rightIndex < items.size()) {  // 如果有右子结点

                    T rightChild = items.get(rightIndex);  // 获
取右子结点的元素对象
                    if (rightChild.compareTo(maxChild) > 0) {  /
/ 找出两个子节点中的最大子结点
                        maxChild = rightChild;
```

```
                    maxIndex = rightIndex;
                }
            }

            // 如果最大子节点比父节点大，则需要向下调整
            if (maxChild.compareTo(intent) > 0) {
                items.set(index, maxChild); // 将子节点向上移
                index = maxIndex; // 记录上移节点的位置
                leftIndex = index * 2 + 1; // 找到上移节点的左
子节点的位置
            } else { // 最大子节点不比父节点大，说明父子路径已经按
从大到小排好顺序了，不需要调整了
                break;
            }
        }

        // index此时记录是的最后一个被上移的子节点的位置（也可能是自
身），所以将最开始的调整的元素值放入index位置即可
        items.set(index, intent);
    }

    /**
     * 向堆中添加一个元素
     *
     * @param item 等待添加的元素
     */
    public void add(T item) {
        items.add(item); // 将元素添加到最后
        siftUp(items.size() - 1); // 循环上移，以完成重构
    }

    /**
     * 删除堆顶元素
     *
     * @return 堆顶部的元素
     */
    public T deleteTop() {
        if (items.isEmpty()) { // 如果堆已经为空，就报出异常
            throw new RuntimeException("The heap is empty.")
;
```

```
            }

            T maxItem = items.get(0); // 获取堆顶元素
            T lastItem = items.remove(items.size() - 1); // 删除
最后一个元素
            if (items.isEmpty()) { // 删除元素后，如果堆为空的情况，
说明删除的元素也是堆顶元素
                return lastItem;
            }

            items.set(0, lastItem); // 将删除的元素放入堆顶
            siftDown(0); // 自上向下调整堆
            return maxItem; // 返回堆顶元素
        }

        /**
         * 获取下一个元素
         *
         * @return 下一个元素对象
         */
        public T next() {

            if (cursor >= items.size()) {
                throw new RuntimeException("No more element");
            }
            return items.get(cursor);

        }

        /**
         * 判断堆中是否还有下一个元素
         *
         * @return true堆中还有下一个元素，false堆中无下五元素
         */
        public boolean hasNext() {
            cursor++;
            return cursor < items.size();
        }

        /**
```

```
         * 获取堆中的第一个元素
         *
         * @return 堆中的第一个元素
         */
        public T first() {
            if (items.size() == 0) {
                throw new RuntimeException("The heap is empty.")
;
            }
            return items.get(0);
        }

        /**
         * 判断堆是否为空
         *
         * @return true是，false否
         */
        public boolean isEmpty() {
            return items.isEmpty();
        }

        /**
         * 获取堆的大小
         *
         * @return 堆的大小
         */
        public int size() {
            return items.size();
        }

        /**
         * 清空堆
         */
        public void clear() {
            items.clear();
        }

        @Override
        public String toString() {
            return items.toString();
```

```
        }
    }

    /**
     * 题目： 输入n个整数，找出其中最小的k个数。
     * 【第二种解法】
     * @param input  输入数组
     * @param output 输出数组
     */
    public static void getLeastNumbers2(int[] input, int[] outpu
t) {
        if (input == null || output == null || output.length <= 0
 || input.length < output.length) {
            throw new IllegalArgumentException("Invalid args");
        }

        MaxHeap<Integer> maxHeap = new MaxHeap<>(output.length);
        for (int i : input) {
            if (maxHeap.size() < output.length) {
                maxHeap.add(i);
            } else {
                int max = maxHeap.first();
                if (max > i) {
                    maxHeap.deleteTop();
                    maxHeap.add(i);
                }
            }
        }

        for (int i = 0; maxHeap.hasNext(); i++) {
            output[i] = maxHeap.next();
        }
    }


    /**
     * 题目： 输入n个整数，找出其中最小的k个数。
     * 【第一种解法】
     * @param input  输入数组
     * @param output 输出数组
```

```
     */
    public static void getLeastNumbers(int[] input, int[] output)
 {

        if (input == null || output == null || output.length <= 0
 || input.length < output.length) {
            throw new IllegalArgumentException("Invalid args");
        }

        int start = 0;
        int end = input.length - 1;
        int index = partition(input, start, end);
        int target = output.length - 1;

        while (index != target) {
            if (index < target) {
                start = index + 1;
            } else {
                end = index - 1;
            }
            index = partition(input, start, end);
        }

        System.arraycopy(input, 0, output, 0, output.length);
    }

    /**
     * 分区算法
     *
     * @param input 输入数组
     * @param start 开始下标
     * @param end   结束下标
     * @return 分区位置
     */
    private static int partition(int[] input, int start, int end)
 {
        int tmp = input[start];

        while (start < end) {
            while (start < end && input[end] >= tmp) {
```

```
                end--;
            }
            input[start] = input[end];

            while (start < end && input[start] <= tmp) {
                start++;
            }
            input[end] = input[start];
        }

        input[start] = tmp;
        return start;
    }
}
```

一、题目

输入一个整型数组，数组里有正数也有负数。数组中一个或连续的多个整数组成一个子数组。求所有子数组的和的最大值。要求时间复杂度为O(n)。

例子说明：

例如输入的数组为{1, -2, 3, 10, -4, 7, 2, -5}，和最大的子数组为｛3, 10, -4, 7, 2}。因此输出为该子数组的和18 。

二、解题思路

解法一：举例分析数组的规律。

我们试着从头到尾逐个累加示例数组中的每个数字。初始化和为0。第一步加上第一个数字1， 此时和为1。接下来第二步加上数字-2，和就变成了-1。第三步刷上数字3。我们注意到由于此前累计的和是－1 ，小于0，那如果用-1 加上3 ，得到的和是2 ， 比3 本身还小。也就是说从第一个数字开始的子数组的和会小于从第三个数字开始的子数组的和。因此我们不用考虑从第一个数字开始的子数组，之前累计的和也被抛弃。

我们从第三个数字重新开始累加，此时得到的和是3 。接下来第四步加10，得到和为13 。第五步加上-4， 和为9。我们发现由于-4 是一个负数，因此累加-4 之后得到的和比原来的和还要小。因此我们要把之前得到的和13 保存下来，它有可能是最大的子数组的和。第六步加上数字7，9 加7 的结果是16，此时和比之前最大的和13 还要大， 把最大的子数组的和由13更新为16。第七步加上2，累加得到的和为18，同时我们也要更新最大子数组的和。第八步加上最后一个数字-5，由于得到的和为13 ，小于此前最大的和18，因此最终最大的子数组的和为18 ，对应的子数组是｛3, 10, -4, 7, 2｝ 。

解法二： 应用动态归划法。

可以用动态规划的思想来分析这个问题。如果用函数f(i)表示以第i个数字结尾的子数组的最大和，那么我们需要求出max[f(i)]，其中0 <= i < n。我们可用如下边归公式求f(i):

$$f(i)=\begin{cases}pData[i] & i=0\text{或者}f(i-1)\le 0\\ f(i-1)+pData[i] & i\ne 0\text{并且}f(i-1)>0\end{cases}$$

这个公式的意义：当以第i-1 个数字结尾的子数组中所有数字的和小于0时，如果把这个负数与第i个数累加，得到的结果比第i个数字本身还要小，所以这种情况下以第i个数字结尾的子数组就是第i个数字本身。如果以第i-1 个数字结尾的子数组中所有数字的和大于0 ,与第i 个数字累加就得到以第i个数字结尾的子数组中所有数字的和。

三、解题代码

```
public class Test {
    /**
     * 输入一个整型数组，数组里有正数也有负数。数组中一个或连
     * 续的多个整数组成一个子数组。求所有子数组的和的最大值。要求时间复杂度
为O(n)。
     *
     * @param arr 输入数组
     * @return 最大的连续子数组和
     */
    public static int findGreatestSumOfSubArray(int[] arr) {
        // 参数校验
        if (arr == null || arr.length < 1) {
            throw new IllegalArgumentException("Array must conta
in an element");
        }

        // 记录最大的子数组和，开始时是最小的整数
        int max = Integer.MIN_VALUE;
        // 当前的和
        int curMax = 0;
        // 数组遍历
        for (int i : arr) {
            // 如果当前和小于等于0，就重新设置当前和
            if (curMax <= 0) {
                curMax = i;
            }
            // 如果当前和大于0，累加当前和
```

```
            else {
                curMax += i;
            }

            // 更新记录到的最在的子数组和
            if (max < curMax) {
                max = curMax;
            }
        }


        return max;
    }
}
```

一、题目

输入一个整数n，求从1 到n这n个整数的十进制表示中1 出现的次数。

举例说明：

例如输入12 ，从1 到12 这些整数中包含1 的数字有1、10、11 和12，1 一共出现了5 次。

二、解题思路

第一种：不考虑时间效率的解法

累加1 到n 中每个整数中1出现的次数。我们可以每次通过对10 求余数判断整数的个位数字是不是1 。如果这个数字大于10，除以10 之后再判断个位数字是不是1 。

第二种：从数字规律着手明显提高时间效率的解法

21345 作为例子来分析。我们把从1 到21345 的所有数字分为两段， 一段是从1 到1345，另一段是从1346 到21345。

我们先看从01346 到21345 中1 出现的次数。1 的出现分为两种情况。首先分析1出现在最高位（本例中是万位）的情况。从01346 到21345 的数字中， 1出现在10000～19999 这10000 个数字的万位中， 一共出现了10000(10^4)个。

值得注意的是， 并不是对所有5 位数而言在万位出现的次数都是10000 个。对于万位是1 的数字比如输入12345, 1 只出现在10000～ 12345 的万位，出现的次数不是10^4 次，而是2346 次，也就是除去最高数字之后剩下的数字再加上1 （即2345+1=2346 次）。

接下来分析1出现在除最高位之外的其他四位数中的情况。例子中01346～21345 这20000 个数字中后4 位中1 出现的次数是2000 次。由于最高位是2，我们可以再把1346～21345 分成两段， 01346～11345 和11346～21345 。每一段剩下的4 位数字中， 选择其中一位是1 ，其余三位可以在0～9 这10 个数字中任意选择，因此根据排列组合原则，总共出现的次数是2*10^3=2000，一共有4位可以选择，所以一共是8000。

至于从1 到1345 中1 出现的次数，我们就可以用递归求得了。这也是我们为什么要把1～21345 分成1～ 1345 和1346～21345 两段的原因。因为把21345 的最高位去掉就变成1345 ，便于我们采用递归的思路。

三、解题代码

```
public class Test {

    /**
     * 题目：输入一个整数n求从1 到n这n个整数的十进制表示中1 出现的次数。
     * @param n 最大的数字
     * @return 1-n中，各个数位1出现的次数
     */
    public static int numberOf1Between1AndN(int n) {
        if (n <= 0) {
            return 0;
        }

        String value = n + "";
        int[] numbers = new int[value.length()];

        for (int i = 0; i < numbers.length; i++) {
            numbers[i] = value.charAt(i) - '0';
        }

        return numberOf1(numbers, 0);
    }

    /**
     * 求0-numbers表的数字中的1的个数
     *
     * @param numbers 数字，如{1, 2, 3, 4, 5}表示数字12345
     * @param curIdx  当前处理的位置
     * @return 1的个数
     */
    private static int numberOf1(int[] numbers, int curIdx) {

        if (numbers == null || curIdx >= numbers.length || curId
```

```
x < 0) {
            return 0;
        }
        // 待处理的第一个数字
        int first = numbers[curIdx];

        // 要处理的数字的位数
        int length = numbers.length - curIdx;

        // 如果只有一位且这一位是0返回0
        if (length == 1 && first == 0) {
            return 0;
        }

        // 如果只有一位且这一位不是0返回1
        if (length == 1 && first > 0) {
            return 1;
        }

        // 假设numbers是21345
        // numFirstDigit是数字10000-19999的第一个位中的数目
        int numFirstDigit = 0;
        // 如果最高位不是1，如21345，在[1236, 21345]中，最高位1出现的只
在[10000, 19999]中，出现1的次数是10^4方个
        if (first > 1) {
            numFirstDigit = powerBase10(length - 1);
        }
        // 如果最高位是1，如12345，在[2346, 12345]中，最高位1出现的只在
[10000, 12345]中，总计2345+1个
        else if (first == 1) {
            numFirstDigit = atoi(numbers, curIdx + 1) + 1;
        }

        // numOtherDigits，是[1346, 21345]中，除了第一位之外（不看213
45中的第一位2）的数位中的1的数目
        int numOtherDigits = first * (length - 1) * powerBase10(
length - 2);
        // numRecursive是1-1234中1的的数目
        int numRecursive = numberOf1(numbers, curIdx + 1);
```

```
        return numFirstDigit + numOtherDigits + numRecursive;
    }

    /**
     * 将数字数组转换成数值，如{1, 2, 3, 4, 5}，i = 2，结果是345
     * @param numbers 数组
     * @param i 开始黑气的位置
     * @return 转换结果
     */
    private static int atoi(int[] numbers, int i) {
        int result = 0;
        for (int j = i; j < numbers.length; j++) {
            result = (result * 10 + numbers[j]);
        }
        return result;
    }

    /**
     * 求10的n次方，假定n不为负数
     * @param n 幂，非负数
     * @return 10的n次方
     */
    private static int powerBase10(int n) {
        int result = 1;
        for (int i = 0; i < n; i++) {
            result *= 10;
        }
        return result;
    }
}
```

一、题目

输入一个正整数数组，把数组里所有数字拼接起来排成一个数，打印能拼接出的所有数字中最小的一个。

例子说明：

例如输入数组{3， 32, 321}，则扫描输出这3 个数字能排成的最小数字321323。

二、解题思路

第一种：直观解法

先求出这个数组中所有数字的全排列，然后把每个排列拼起来，最后求出拼起来的数字的最小值。

第二种：排序解法

找到一个排序规则，数组根据这个规则排序之后能排成一个最小的数字。要确定排序规则，就要比较两个数字，也就是给出两个数字m 和n，我们需要确定一个规则判断m 和n 哪个应该排在前面，而不是仅仅比较这两个数字的值哪个更大。

根据题目的要求，两个数字m 和n能拼接成数字m和m。如果mn < nm，那么我们应该打印出m，也就是m 应该排在n 的前面，我们定义此时m 小于n：反之，如果nm < mn，我们定义n小于m。如果mn=nm,m 等于n。在下文中，符号“<”、“>”及“＝”表示常规意义的数值的大小关系，而文字“大于”、“小于”、“等于”表示我们新定义的大小关系。

接下来考虑怎么去拼接数字，即给出数字m和n，怎么得到数字m和m 并比较它们的大小。直接用数值去计算不难办到，但需要考虑到一个潜在的问题就是m 和n 都在int 能表达的范围内，但把它们拼起来的数字mn 和nm 用int 表示就有可能溢出了，所以这还是一个隐形的大数问题。

一个非常直观的解决大数问题的方法就是把数字转换成字符串。另外，由于把数字m 和n 拼接起来得到mn 和nm，它们的位数肯定是相同的，因此比较它们的大小只需要按照字符串大小的比较规则就可以了。

三、解题代码

```
public class Test {

    /**
     * 自定义的排序比较器，实现算法说明的排序原理
     */
    private static class MComparator implements Comparator<String
> {

        @Override
        public int compare(String o1, String o2) {

            if (o1 == null || o2 == null) {
                throw new IllegalArgumentException("Arg should n
ot be null");
            }

            String s1 = o1 + o2;
            String s2 = o2 + o1;
            return s1.compareTo(s2);
        }
    }

    /**
     * 快速排序算法
     *
     * @param array      待排序数组
     * @param start      要排序的起始位置
     * @param end        要排序的结束位置
     * @param comparator 自定义的比较器
     */
    private static void quickSort(String[] array, int start, int
 end, Comparator<String> comparator) {

        if (start < end) {
            String pivot = array[start];
            int left = start;
            int right = end;
            while (start < end) {
                while (start < end && comparator.compare(array[e
```

```
nd], pivot) >= 0) {
                    end--;
                }

                array[start] = array[end];

                while (start < end && comparator.compare(array[s
tart], pivot) <= 0) {
                    start++;
                }
                array[end] = array[start];

            }

            array[start] = pivot;

            quickSort(array, left, start - 1, comparator);
            quickSort(array, start + 1, end, comparator);
        }
    }

    /**
     * 题目：输入一个正整数数组，把数组里所有数字拼接起来排成一个数，
     * 打印能拼接出的所有数字中最小的一个。
     * @param array 输入的数组
     * @return 输出结果
     */
    public static String printMinNumber(String[] array) {

        if (array == null || array.length < 1) {
            throw new IllegalArgumentException("Array must conta
in value");
        }

        MComparator comparator = new MComparator();
        quickSort(array, 0, array.length - 1, comparator);

        StringBuilder builder = new StringBuilder(256);
        for (String s : array) {
            builder.append(s);
```

```
        }

        return builder.toString();
    }
}
```

一、题目

我们把只包含因子2、3 和5 的数称作丑数（Ugly Number）。求从小到大的顺序的第1500个丑数。

举例说明：

例如6、8 都是丑数，但14 不是，它包含因子7。习惯上我们把1 当做第一个丑数。

二、解题思路

第一种：逐个判断每个数字是不是丑数的解法，直观但不够高效。

第二种：创建数组保存已经找到丑数，用空间换时间的解法。

根据丑数的定义， 丑数应该是另一个丑数乘以2、3 或者5 的结果（1除外）。因此我们可以创建一个数组，里面的数字是排好序的丑数，每一个丑数都是前面的丑数乘以2、3或者5得到的。

这种思路的关键在于怎样确保数组里面的丑数是排好序的。假设数组中已经有若干个丑数排好序后存放在数组中，并且把已有最大的丑数记做M，我们接下来分析如何生成下一个丑数。该丑数肯定是前面某一个丑数乘以2、3 或者5 的结果， 所以我们首先考虑把已有的每个丑数乘以2。在乘以2 的时候能得到若干个小于或等于M 的结果。由于是按照顺序生成的，小于或者等于M 肯定已经在数组中了，我们不需再次考虑：还会得到若干个大于M 的结果，但我们只需要第一个大于M 的结果，因为我们希望丑数是按从小到大的顺序生成的，其他更大的结果以后再说。我们把得到的第一个乘以2 后大于M 的结果记为M2，同样，我们把已有的每一个丑数乘以3 和5，能得到第一个大于M 的结果M3 和M5，那么下一个丑数应该是M2、M3 和M5 这3个数的最小者。

前面分析的时候，提到把已有的每个丑数分别都乘以2、3 和5。事实上这不是必须的，因为已有的丑数是按顺序存放在数组中的。对乘以2而言， 肯定存在某一个丑数T2，排在它之前的每一个丑数乘以2 得到的结果都会小于已有最大的丑数，在它之后的每一个丑数乘以2 得到的结果都会太大。我们只需记下这个丑数的位置， 同时每次生成新的丑数的时候，去更新这个T2。对乘以3 和5 而言， 也存在着同样的T3和T5。

三、解题代码

```
public class Test {
    /**
     * 判断一个数是否只有2，3，5因子（丑数）
     *
     * @param num 待判断的数，非负
     * @return true是丑数，false丑数
     */
    private static boolean isUgly(int num) {
        while (num % 2 == 0) {
            num /= 2;
        }

        while (num % 3 == 0) {
            num /= 3;
        }

        while (num % 5 == 0) {
            num /= 5;
        }

        return num == 1;
    }

    /**
     * 找第index个丑数，速度太慢
     *
     * @param index 第index个丑数
     * @return 对应的丑数值
     */
    public static int getUglyNumber(int index) {
        if (index <= 0) {
            return 0;
        }

        int num = 0;
        int uglyFound = 0;
        while (uglyFound < index) {
```

```
            num++;
            if (isUgly(num)) {
                ++uglyFound;
            }
        }

        return num;
    }

    /**
     * 找第index个丑数，【第二种方法】
     *
     * @param index 第index个丑数
     * @return 对应的丑数值
     */
    public static int getUglyNumber2(int index) {
        if (index <= 0) {
            return 0;
        }

        int[] pUglyNumbers = new int[index];
        pUglyNumbers[0] = 1;
        int nextUglyIndex = 1;

        int p2 = 0;
        int p3 = 0;
        int p5 = 0;

        while (nextUglyIndex < index) {
            int min = min(pUglyNumbers[p2] * 2, pUglyNumbers[p3]
 * 3, pUglyNumbers[p5] * 5);
            pUglyNumbers[nextUglyIndex] = min;

            while (pUglyNumbers[p2] * 2 <= pUglyNumbers[nextUgly
Index]) {
                p2++;
            }

            while (pUglyNumbers[p3] * 3 <= pUglyNumbers[nextUgly
Index]) {
```

```
                p3++;
            }

            while (pUglyNumbers[p5] * 5 <= pUglyNumbers[nextUgly
Index]) {
                p5++;
            }

            nextUglyIndex++;
        }

        return pUglyNumbers[nextUglyIndex - 1];
    }

    private static int min(int n1, int n2, int n3) {
        int min = n1 < n2 ? n1 : n2;
        return min < n3 ? min : n3;
    }
}
```

一、题目

在字符串中找出第一个只出现一次的字符。

二、解题思路

第一种：直接求解：

从头开始扫描这个字符串中的每个字符。当访问到某字符时拿这个字符和后面的每个字符相比较，如果在后面没有发现重复的字符，则该字符就是只出现一次的字符。如果字符串有n个字符，每个字符可能与后面的O(n）个字符相比较，因此这种思路的时间复杂度是O(n^2)。

第二种：记录法

由于题目与字符出现的次数相关， 我们是不是可以统计每个字符在该字符串中出现的次数？要达到这个目的，我们需要一个数据容器来存放每个字符的出现次数。在这个数据容器中可以根据字符来查找它出现的次数，也就是说这个容器的作用是把一个字符映射成二个数字。在常用的数据容器中， 哈希表正是这个用途。 为了解决这个问题，我们可以定义哈希表的键（Key）是字符， 而值(Value ）是该字符出现的次数。同时我们还需要从头开始扫描字符串两次。第一次扫描字符串时，每扫描到一个字符就在哈希表的对应项中把次数加1 。接下来第二次扫描时， 每扫描到一个字符就能从哈希表中得到该字符出现的次数。这样第一个只出现一次的字符就是符合要求的输出。 第一次扫描时，在哈希表中更新一个字符出现的次数的时间是O(n) 。如果字符串长度为n， 那么第一次扫描的时间复杂度是O(n)。第二次扫描时，同样0(1)能读出一个字符出现的次数，所以时间复杂度仍然是O(n)。这样算起来，总的时间复杂度是O(n)。

三、解题代码

```
public class Test {
    public static char firstNotRepeatingChar(String s) {
        if (s == null || s.length() < 1) {
            throw new IllegalArgumentException("Arg should not b
e null or empty");
        }

        Map<Character, Integer> map = new LinkedHashMap<>();
        for (int i = 0; i < s.length(); i++) {
            char c = s.charAt(i);
            if (map.containsKey(c)) {
                map.put(c, -2);
            } else {
                map.put(c, i);
            }
        }

        Set<Map.Entry<Character, Integer>> entrySet = map.entryS
et();
        // 记录只出现一次的字符的索引
        int idx = Integer.MAX_VALUE;
        // 记录只出现一次的字符
        char result = '\0';

        // 找最小索引对应的字符
        for (Map.Entry<Character, Integer> entry : entrySet) {
            if (entry.getValue() >= 0 && entry.getValue() < idx)
 {
                idx = entry.getValue();
                result = entry.getKey();
            }
        }

        return result;
    }
}
```

一、题目

在数组中的两个数字如果前面一个数字大于后面的数字，则这两个数字组成一个逆序对。输入一个数组，求出这个数组中的逆序对的总数。

举例分析

例如在数组｛7, 5, 6, 4 中， 一共存在5 个逆序对，分别是（7, 6）、（7，5），(7, 4）、（6, 4）和（5, 4）。

二、解题思路

第一种：直接求解

顺序扫描整个数组。每扫描到一个数字的时候，逐个比较该数字和它后面的数字的大小。如果后面的数字比它小，则这两个数字就组成了一个逆序对。假设数组中含有n 个数字。由于每个数字都要和O(n）个数字作比较， 因此这个算法的时间复杂度是O(n^2)。

第二种：分析法

我们以数组｛7, 5, 6, 4｝为例来分析统计逆序对的过程。每次扫描到一个数字的时候，我们不能拿它和后面的每一个数字作比较，否则时间复杂度就是O(n^2)，因此我们可以考虑先比较两个相邻的数字。

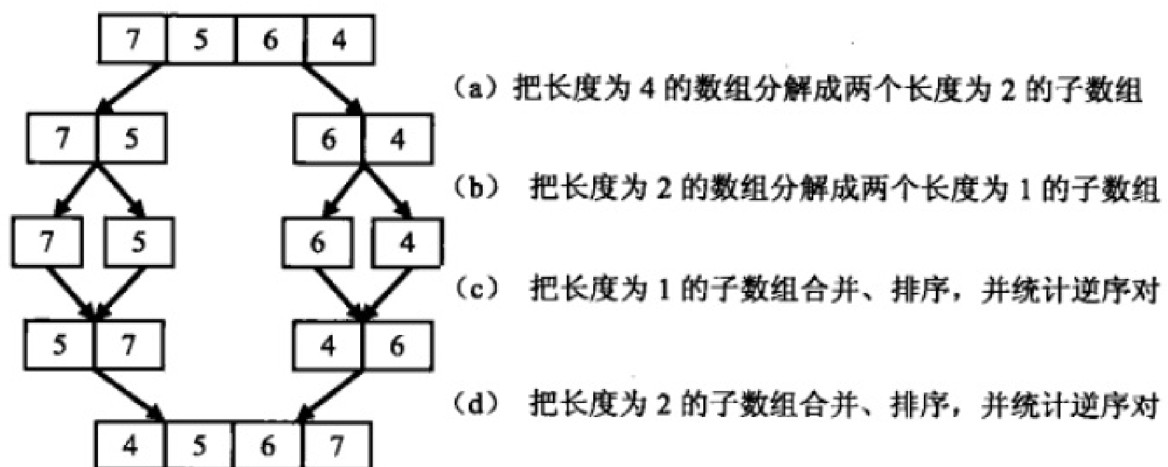


图 5.1　统计数组{7, 5, 6, 4} 中逆序对的过程

如图5．1 (a)和图5.1 (b）所示，我们先把数组分解成两个长度为2的子数组，再把这两个子数组分别拆分成两个长度为1 的子数组。接下来一边合并相邻的子数组， 一边统计逆序对的数目。在第一对长度为1 的子数组｛7｝、｛5｝中7 大于5 ， 因此（7, 5）组成一个逆序对。同样在第二对长度为1 的子数组｛6｝、｛4｝中也有逆序对（6, 4）。由于我们已经统计了这两对子数组的逆序对，因此需要把这两对子数组排序（ 图5.1 (c）所示），以免在以后的统计过程中再重复统计。

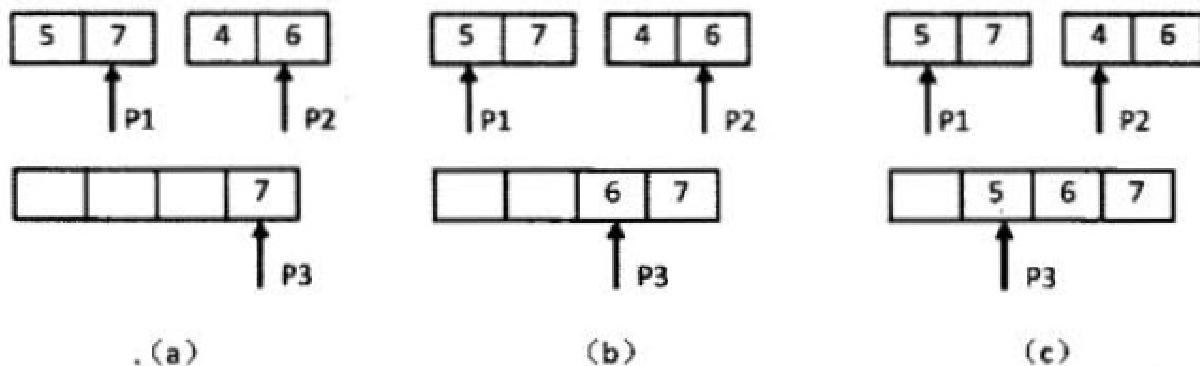


图 5.2　图 5.1（d）中合并两个子数组并统计逆序对的过程

> 注　图中省略了最后一步， 即复制第二个子数组最后剩余的4 到辅助数组中. (a) P1指向的数字大于P2指向的数字，表明数组中存在逆序对 · P2 指向的数字是第二个子数组的第二个数字， 因此第二个子数组中有两个数字比7 小 · 把逆序对数目加2，并把7 复制到辅助数组，向前移动P1和P3. (b) P1指向的数字小子P2 指向的数字，没有逆序对 · 把P2 指向的数字复制到辅助数组，并向前移动P2 和P3 . (c) P1指向的数字大于P2 指向的数字，因此存在逆序对 · 由于P2 指向的数字是第二个子数组的第一个数字，子数组中只有一个数字比5 小 · 把逆序对数目加1 ，并把5复制到辅助数组，向前移动P1和P3 .

接下来我们统计两个长度为2 的子数组之间的逆序对。我们在图5.2 中细分图5.1 (d）的合并子数组及统计逆序对的过程。

我们先用两个指针分别指向两个子数组的末尾，并每次比较两个指针指向的数字。如果第一个子数组中的数字大于第二个子数组中的数字，则构成逆序对，并且逆序对的数目等于第二个子数组中剩余数字的个数（如图5.2 (a)和图5.2 (c)所示）。如果第一个数组中的数字小于或等于第二个数组中的数字，则不构成逆序对（如图5.2 (b)所示〉。每一次比较的时候，我们都把较大的数字从后往前复制到一个辅助数组中去，确保辅助数组中的数字是递增排序的。在把较大的数字复制到辅助数组之后，把对应的指针向前移动一位，接下来进行下一轮比较。

经过前面详细的诗论， 我们可以总结出统计逆序对的过程：先把数组分隔成子数组，然后再统计出两个相邻子数组之间的逆序对的数目 。在统计逆序对的过程中，还需要对数组进行排序。如果对排序算法很熟悉，我们不难发现这个排序的过程实际上就是归并排序。

三、解题代码

```
public class Test {

    public static int inversePairs(int[] data) {
        if (data == null || data.length < 1) {
            throw new IllegalArgumentException("Array arg should
 contain at least a value");
        }

        int[] copy = new int[data.length];
        System.arraycopy(data, 0, copy, 0, data.length);

        return inversePairsCore(data, copy, 0, data.length - 1);
    }

    private static int inversePairsCore(int[] data, int[] copy,
int start, int end) {

        if (start == end) {
            copy[start] = data[start];
            return 0;
        }

        int length = (end - start) / 2;
        int left = inversePairsCore(copy, data, start, start + l
ength);
        int right = inversePairsCore(copy, data, start + length
+ 1, end);

        // 前半段的最后一个数字的下标
        int i = start + length;
        // 后半段最后一个数字的下标
```

```
        int j = end;
        // 开始拷贝的位置
        int indexCopy = end;
        // 逆序数
        int count = 0;

        while (i >= start && j >= start + length + 1) {
            if (data[i] > data[j]) {
                copy[indexCopy] = data[i];
                indexCopy--;
                i--;
                count += j - (start + length); // 对应的逆序数
            } else {
                copy[indexCopy] = data[j];
                indexCopy--;
                j--;
            }
        }

        for (; i >= start;) {
            copy[indexCopy] = data[i];
            indexCopy--;
            i--;
        }

        for (; j >= start + length + 1;) {
            copy[indexCopy] = data[j];
            indexCopy--;
            j--;
        }
        return count + left + right;
    }
}
```

一、题目

输入两个链表，找出它们的第一个公共结点。

二、解题思路

第一种：直接法 在第一个链表上顺序遍历每个结点，每遍历到一个结点的时候，在第二个链表上顺序遍历每个结点。如果在第二个链表上有一个结点和第一个链表上的结点一样，说明两个链表在这个结点上重合，于是就找到了它们的公共结点。如果第一个链表的长度为m，第二个链表的长度为n，显然该方法的时间复杂度是O(mn）。

第二种：使用栈 所以两个有公共结点而部分重舍的链衰，拓扑形状看起来像一个Y， 而不可能像X（如图5.3 所示）。

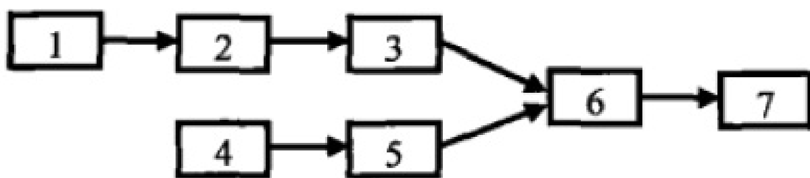


图 5.3　两个链表在值为 6 的结点处交汇

经过分析我们发现，如果两个链表有公共结点，那么公共结点出现在两个链表的尾部。如果我们从两个链衰的尾部开始往前比较，最后一个相同的结点就是我们要找的结点。 在上述思路中，我们需要用两个辅助钱。如果链表的长度分别为m 和n，那么空间复杂度是O(m+n）。这种思路的时间复杂度也是O(m+n）。和最开始的蛮力法相比，时间效率得到了提高，相当于是用空间消耗换取了时间效率。

第三种：先行法 在图5.3 的两个链表中，我们可以先遍历一次得到它们的长度分别为5 和4， 也就是较长的链表与较短的链表相比多一个结点。第二次先在长的链表上走1 步，到达结点2。接下来分别从结点2 和结点4 出发同时遍历两个结点， 直到找到它们第一个相同的结点6，这就是我们想要的结果。 第三种思路和第二种思路相比，时间复杂度都一样， 但我们不再需要辅助的拢，因此提高了空间效率。

三、解题代码

```
public class Test {
    /**
     * 链表结点类
     */
    private static class ListNode {
        int val;
        ListNode next;

        public ListNode() {

        }

        public ListNode(int val) {
            this.val = val;
        }

        @Override
        public String toString() {
            return val + "";
        }
    }

    /**
     * 找两个结点的第一个公共结点，如果没有找到返回null，方法比较好，考虑了
两个链表中有null的情况
     *
     * @param head1 第一个链表
     * @param head2 第二个链表
     * @return 找到的公共结点，没有返回null
     */
    public static ListNode findFirstCommonNode(ListNode head1, L
istNode head2) {
        int length1 = getListLength(head1);
        int length2 = getListLength(head2);

        int diff = length1 - length2;
        ListNode longListHead = head1;
        ListNode shortListHead = head2;

        if (diff < 0) {
```

```
            longListHead = head2;
            shortListHead = head1;
            diff = length2 - length1;
        }

        for (int i = 0; i < diff; i++) {
            longListHead = longListHead.next;
        }

        while (longListHead != null && shortListHead != null &&
longListHead != shortListHead) {
            longListHead = longListHead.next;
            shortListHead = shortListHead.next;
        }

        // 返回第一个相同的公共结点，如果没有返回null
        return longListHead;
    }

    private static int getListLength(ListNode head) {
        int result = 0;
        while (head != null) {
            result++;
            head = head.next;
        }

        return result;
    }
}
```

一、题目

统计一个数字：在排序数组中出现的次数。

举例说明

例如输入排序数组｛ 1, 2, 3, 3, 3, 3, 4, 5｝和数字3 ，由于3 在这个数组中出现了4次，因此输出4 。

二、解题思路

利用改进的二分算法。

如何用二分查找算法在数组中找到第一个k，二分查找算法总是先拿数组中间的数字和k作比较。如果中间的数字比k大，那么k只有可能出现在数组的前半段，下一轮我们只在数组的前半段查找就可以了。如果中间的数字比k小，那么k只有可能出现在数组的后半段，下一轮我们只在数组的后半乓查找就可以了。如果中间的数字和k 相等呢？我们先判断这个数字是不是第一个k。如果位于中间数字的前面一个数字不是k,此时中间的数字刚好就是第一个k。如果中间数字的前面一个数字也是k，也就是说第一个k肯定在数组的前半段， 下一轮我们仍然需要在数组的前半段查找。

同样的思路在排序数组中找到最后一个k。如果中间数字比k大，那么k只能出现在数组的前半段。如果中间数字比k小，k就只能出现在数组的后半段。如果中间数字等于k呢？我们需要判断这个k是不是最后一个k，也就是中间数字的下一个数字是不是也等于k。如果下一个数字不是k，则中间数字就是最后一个k了：否则下一轮我们还是要在数组的后半段中去查找。

三、解题代码

```
public class Test {
    /**
     * 找排序数组中k第一次出现的位置
     *
     * @param data
     * @param k
```

```
     * @param start
     * @param end
     * @return
     */
    private static int getFirstK(int[] data, int k, int start, i
nt end) {
        if (data == null || data.length < 1 || start > end) {
            return -1;
        }

        int midIdx = start + (end - start) / 2;
        int midData = data[midIdx];

        if (midData == k) {
            if (midIdx > 0 && data[midIdx - 1] != k || midIdx ==
0) {
                return midIdx;
            } else {
                end = midIdx - 1;
            }
        } else if (midData > k) {
            end = midIdx - 1;
        } else {
            start = midIdx + 1;
        }

        return getFirstK(data, k, start, end);
    }

    /**
     * 找排序数组中k最后一次出现的位置
     *
     * @param data
     * @param k
     * @param start
     * @param end
     * @return
     */
    private static int getLastK(int[] data, int k, int start, int
 end) {
```

```
        if (data == null || data.length < 1 || start > end) {
            return -1;
        }

        int midIdx = start + (end - start) / 2;
        int midData = data[midIdx];

        if (midData == k) {
            if (midIdx + 1 < data.length && data[midIdx + 1] !=
k || midIdx == data.length - 1) {
                return midIdx;
            } else {
                start = midIdx + 1;
            }
        } else if (midData < k) {
            start = midIdx + 1;
        } else {
            end = midIdx - 1;
        }

        return getLastK(data, k, start, end);
    }

    /**
     * 题目：统计一个数字：在排序数组中出现的次数
     * @param data
     * @param k
     * @return
     */
    public static int getNumberOfK(int[] data, int k) {
        int number = 0;
        if (data != null && data.length > 0) {
            int first = getFirstK(data, k, 0, data.length - 1);
            int last = getLastK(data, k, 0, data.length - 1);

            if (first > -1 && last > -1) {
                number = last - first + 1;
            }
        }
```

```
            return number;
        }
    }
```

一、题目

输入一棵二叉树的根结点，求该树的深度。从根结点到叶子点依次经过的结点（含根、叶结点）形成树的一条路径，最长路径的长度为树的深度。

二、解题思路

如果一棵树只有一个结点，它的深度为1。 如果根结点只有左子树而没有右子树，那么树的深度应该是其左子树的深度加1，同样如果根结点只有右子树而没有左子树，那么树的深度应该是其右子树的深度加1. 如果既有右子树又有左子树， 那该树的深度就是其左、右子树深度的较大值再加1。

三、解题代码

```
public static int treeDepth(BinaryTreeNode root) {
    if (root == null) {
        return 0;
    }

    int left = treeDepth(root.left);
    int right = treeDepth(root.right);

    return left > right ? (left + 1) : (right + 1);
}
```

一、题目

输入一棵二叉树的根结点，判断该树是不是平衡二叉树。如果某二叉树中任意结点的左右子树的深度相差不超过1 ，那么它就是一棵平衡二叉树。

二、解题思路

解法一：需要重蟹遍历结点多次的解法 在遍历树的每个结点的时候，调用函数treeDepth得到它的左右子树的深度。如果每个结点的左右子树的深度相差都不超过1 ，按照定义它就是一棵平衡的二叉树。

解法二：每个结点只遍历一次的解法 用后序遍历的方式遍历二叉树的每一个结点，在遍历到一个结点之前我们就已经遍历了它的左右子树。只要在遍历每个结点的时候记录它的深度（某一结点的深度等于它到叶节点的路径的长度），我们就可以一边遍历一边判断每个结点是不是平衡的。

三、解题代码

```
public class Test {

    private static class BinaryTreeNode {
        int val;
        BinaryTreeNode left;
        BinaryTreeNode right;

        public BinaryTreeNode() {
        }

        public BinaryTreeNode(int val) {
            this.val = val;
        }
    }

    public static int treeDepth(BinaryTreeNode root) {
        if (root == null) {
            return 0;
```

```
        }

        int left = treeDepth(root.left);
        int right = treeDepth(root.right);

        return left > right ? (left + 1) : (right + 1);
    }

    /**
     * 判断是否是平衡二叉树，第一种解法
     *
     * @param root
     * @return
     */
    public static boolean isBalanced(BinaryTreeNode root) {
        if (root == null) {
            return true;
        }

        int left = treeDepth(root.left);
        int right = treeDepth(root.right);
        int diff = left - right;
        if (diff > 1 || diff < -1) {
            return false;
        }

        return isBalanced(root.left) && isBalanced(root.right);
    }


    /**
     * 判断是否是平衡二叉树，第二种解法
     *
     * @param root
     * @return
     */
    public static boolean isBalanced2(BinaryTreeNode root) {
        int[] depth = new int[1];
        return isBalancedHelper(root, depth);
    }
```

```
    public static boolean isBalancedHelper(BinaryTreeNode root,
int[] depth) {
        if (root == null) {
            depth[0] = 0;
            return true;
        }

        int[] left = new int[1];
        int[] right = new int[1];

        if (isBalancedHelper(root.left, left) && isBalancedHelpe
r(root.right, right)) {
            int diff = left[0] - right[0];
            if (diff >= -1 && diff <= 1) {
                depth[0] = 1 + (left[0] > right[0] ? left[0] : r
ight[0]);
                return true;
            }
        }

        return false;
    }
}
```

一、题目

一个整型数组里除了两个数字之外，其他的数字都出现了两次，请写程序找出这两个只出现一次的数字。要求时间复杂度是O(n)，空间复杂度是O(1)。

举例说明

例如输入数组{2, 4, 3, 6, 3, 2, 5}，因为只有4 、6 这两个数字只出现一次，其他数字都出现了两次，所以输出4和6 。

二、解题思路

这两个题目都在强调一个（或两个）数字只出现一次，其他的出现两次。这有什么意义呢？我们想到异或运算的一个性质：任何一个数字异或它自己都等于0。也就是说， 如果我们从头到尾依次异或数组中的每一个数字，那么最终的结果刚好是那个只出现一次的数字，因为那些成对出现两次的数字全部在异或中抵消了。

想明白怎么解决这个简单问题之后，我们再回到原始的问题，看看能不能运用相同的思路。我们试着把原数组分成两个子数组，使得每个子数组包含一个只出现一次的数字，而其他数字都成对出现两次。如果能够这样拆分成两个数组， 我们就可以按照前面的办法分别找出两个只出现一次的数字了。

我们还是从头到尾依次异或数组中的每一个数字，那么最终得到的结果就是两个只出现一次的数字的异或结果。因为其他数字都出现了两次，在异或中全部抵消了。由于这两个数字肯定不一样，那么异或的结果肯定不为0，也就是说在这个结果数字的二进制表示中至少就有一位为1 。我们在结果数字中找到第一个为1 的位的位置，记为第n 位。现在我们以第n位是不是１为标准把原数组中的数字分成两个子数组，第一个子数组中每个数字的第n 位都是1 ， 而第二个子数组中每个数字的第n 位都是0。由于我们分组的标准是数字中的某一位是1 还是0 ， 那么出现了两次的数字肯定被分配到同一个子数组。因为两个相同的数字的任意一位都是相同的，我们不可能把两个相同的数字分配到两个子数组中去，于是我们已经把原数组分成了两个子数组，每个子数组都包含一个只出现一次的数字，而其他数字都出现了两次。我们已经知道如何在数组中找出唯一一个只出现一次数字， 因此到此为止所有的问题都已经解决了。

三、解题代码

```
public class Test {
    public static int[] findNumbersAppearanceOnce(int[] data) {
        int[] result = {0, 0};

        if (data == null || data.length < 2) {
            return result;
        }

        int xor = 0;
        for (int i : data) {
            xor ^= i;
        }

        int indexOf1 = findFirstBit1(xor);

        for (int i : data) {
            if (isBit1(i, indexOf1)) {
                result[0] ^= i;
            } else {
                result[1] ^= i;
            }
        }

        return result;
    }

    private static int findFirstBit1(int num) {
        int index = 0;
        while ((num & 1) == 0 && index < 32) {
            num >>>= 1;
            index++;
        }

        return index;
    }

    private static boolean isBit1(int num, int indexBit) {
        num >>>= indexBit;
        return (num & 1) == 1;
```

```
    }
}
```

一、题目

输入一个递增排序的数组和一个数字s，在数组中查找两个数，得它们的和正好是s。如果有多对数字的和等于s，输出任意一对即可。

举例说明

例如输入数组｛1 、2 、4、7 、11 、15 ｝和数字15. 由于4+ 11 = 15 ，因此输出4和11 。

二、解题思路

我们先在数组中选择两个数字，如果它们的和等于输入的s，我们就找到了要找的两个数字。如果和小于s 呢？我们希望两个数字的和再大一点。由于数组已经排好序了，我们可以考虑选择较小的数字后面的数字。因为排在后面的数字要大一些，那么两个数字的和也要大一些， 就有可能等于输入的数字s 了。同样， 当两个数字的和大于输入的数字的时候，我们可以选择较大数字前面的数字，因为排在数组前面的数字要小一些。

三、解题代码

```
/**
 * 输入一个递增排序的数组和一个数字s，在数组中查找两个数，使得得它们的和正好
是s。
 * 如果有多对数字的和等于s，输出任意一对即可。
 *
 * @param data
 * @param sum
 * @return
 */
public static List<Integer> findNumbersWithSum(int[] data, int s
um) {
    List<Integer> result = new ArrayList<>(2);

    if (data == null || data.length < 2) {
        return result;
    }

    int ahead = data.length - 1;
    int behind = 0;
    long curSum; // 统计和，取long是防止结果溢出

    while (behind < ahead) {
        curSum = data[behind] + data[ahead];

        if (curSum == sum) {
            result.add(data[behind]);
            result.add(data[ahead]);
            break;
        } else if (curSum < sum) {
            behind++;
        } else {
            ahead--;
        }
    }

    return result;
}
```

时间： 2018-05-12 01:44:07

一、题目

输入一个正数s，打印出所有和为s 的连续正数序列（至少两个数）。

举例说明

例如输入15，由于1+2+3+4+5=4＋5+6＝7+8=15，所以结果打出3 个连续序列1～5、4～6 和7～8。

二、解题思路

考虑用两个数small 和big 分别表示序列的最小值和最大值。首先把small 初始化为1, big 初始化为2。如果从small 到big 的序列的和大于s，我们可以从序列中去掉较小的值，也就是增大small 的值。如果从small 到big 的序列的和小于s，我们可以增大big，让这个序列包含更多的数字。因为这个序列至少要有两个数字，我们一直增加small 到(1+s)/2 为止。

以求和为9 的所有连续序列为例，我们先把small 初始化为1, big 初始化为2。此时介于small 和big 之间的序列是｛1，2}，序列的和为3，小于9，所以我们下一步要让序列包含更多的数字。我们把big 增加1 变成3,此时序列为{1，2，3}。由于序列的和是6，仍然小于9，我们接下来再增加big 变成4，介于small 和big 之间的序列也随之变成｛ l, 2, 3, 4｝ 。由于列的和10 大于9，我们要删去去序列中的一些数字， 于是我们增加small 变成2，此时得到的序列是｛2, 3, 4｝ ， 序列的和E好是9。我们找到了第一个和为9 的连续序列，把它打印出来。接下来我们再增加big，重复前面的过程，可以找到第二个和为9 的连续序列｛4，5}。

三、解题代码

```
public static List<List<Integer>> findContinuousSequence(int sum
) {
    List<List<Integer>> result = new ArrayList<>();
    if (sum < 3) {
        return result;
    }

    int small = 1;
```

```
        int big = 2;
        int middle = (1 + sum) / 2;
        int curSum = small + big;

        while (small < middle) {
            if (curSum == sum) {
                List<Integer> list = new ArrayList<>(2);
                for (int i = small; i <= big; i++) {
                    list.add(i);
                }
                result.add(list);
            }

            while (curSum > sum && small < middle) {
                curSum -= small;
                small++;

                if (curSum == sum) {
                    List<Integer> list = new ArrayList<>(2);
                    for (int i = small; i <= big; i++) {
                        list.add(i);
                    }
                    result.add(list);
                }
            }

            big++;
            curSum += big;
        }

        return result;
    }
```

一、题目

输入一个英文句子，翻转句子中单词的顺序，但单词内字的顺序不变。为简单起见，标点符号和普通字母一样处理。

举例说明

例如输入字符串”I am a student. ”，则输出”student. a am I”。

二、解题思路

第一步翻转句子中所有的字符。比如翻转“I am a student. ”中所有的字符得到”.tneduts a m a I”，此时不但翻转了句子中单词的顺序，连单词内的字符顺序也被翻转了。第二步再翻转每个单词中字符的顺序，就得到了”student. a am I”。这正是符合题目要求的输出。

三、解题代码

```
/**
 * 题目一：输入一个英文句子，翻转句子中单词的顺序，但单词内字啊的顺序不变。
 * 为简单起见，标点符号和普通字母一样处理。
 *
 * @param data
 * @return
 */
public static char[] reverseSentence(char[] data) {
    if (data == null || data.length < 1) {
        return data;
    }

    reverse(data, 0, data.length - 1);

    int start = 0;
    int end = 0;
```

```
    while (start < data.length) {
        if (data[start] == ' ') {
            start++;
            end++;
        } else if (end == data.length || data[end] == ' ') {
            reverse(data, start, end - 1);
            end++;
            start = end;
        } else {
            end++;
        }
    }

    return data;
}
/**
 * 将data中start到end之间的数字反转
 *
 * @param data
 * @param start
 * @param end
 */
public static void reverse(char[] data, int start, int end) {
    if (data == null || data.length < 1 || start < 0 || end > da
ta.length || start > end) {
        return;
    }

    while (start < end) {
        char tmp = data[start];
        data[start] = data[end];
        data[end] = tmp;

        start++;
        end--;
    }
}
```

时间： 2018-05-12 01:44:07

一、题目

字符串的左旋转操作是把字符串前面的若干个字符转移到字符串的尾部。请定义一个函数实现字符串左旋转操作的功能。

举例说明

比如输入字符串"abcdefg"和数字2，该函数将返回左旋转2位得到的结"cdefgab"。

二、解题思路

以"abcdefg"为例，我们可以把它分为两部分。由于想把它的前两个字符移到后面，我们就把前两个字符分到第一部分，把后面的所有字符都分到第二部分。我们先分别翻转这两部分，于是就得到"bagfedc"。接下来我们再翻转整个字符串，得到的"cdefgab"就是把原始字符串左旋转2位的结果。

三、解题代码

```
/**
 * 题目二：字符串的左旋转操作是把字符串前面的若干个字符转移到字符串的尾部。
 * 请定义一个函数实现字符串左旋转操作的功能。
 * @param data
 * @param n
 * @return
 */
public static char[] leftRotateString(char[] data, int n) {
    if (data == null || n < 0 || n > data.length) {
        return data;
    }

    reverse(data, 0, data.length - 1);
    reverse(data, 0, data.length - n - 1);
    reverse(data, data.length - n, data.length - 1);

    return data;
}
```

```
/**
 * 将data中start到end之间的数字反转
 *
 * @param data
 * @param start
 * @param end
 */
public static void reverse(char[] data, int start, int end) {
    if (data == null || data.length < 1 || start < 0 || end > da
ta.length || start > end) {
        return;
    }

    while (start < end) {
        char tmp = data[start];
        data[start] = data[end];
        data[end] = tmp;

        start++;
        end--;
    }
}
```

一、题目

把n个骰子扔在地上，所有骰子朝上一面的点数之和为s。输入n，打印出s 的所有可能的值出现的概率。

二、解题思路

解法一：基于通归求解，时间效率不够高。

先把n个骰子分为两堆：第一堆只有一个，另一个有n- 1 个。单独的那一个有可能出现从1 到6 的点数。我们需要计算从1 到6 的每一种点数和剩下的n-1 个骰子来计算点数和。接下来把剩下的n-1个骰子还是分成两堆，第一堆只有一个， 第二堆有n-2 个。我们把上一轮那个单独骰子的点数和这一轮单独骰子的点数相加， 再和剩下的n-2 个骰子来计算点数和。分析到这里，我们不难发现这是一种递归的思路，递归结束的条件就是最后只剩下一个骰子。

我们可以定义一个长度为6n-n+1 的数组， 和为s 的点数出现的次数保存到数组第s-n个元素里。

解法二：基于循环求解，时间性能好

我们可以考虑用二维数组来存储骰子点数的每一个总数出现的次数。在一次循环中， 第一个数组中的第n 个数字表示骰子和为n 出现的次数。在下一循环中，我们加上一个新的骰子，此时和为n 的骰子出现的次数应该等于上一次循环中骰子点数和为n-1 、n-2 、n-3 、n-4, n-5 与n-6 的次数的总和，所以我们把另一个数组的第n个数字设为前一个数组对应的第n-1 、n-2 、n-3 、n-4、n-5与n-6之和。

三、解题代码

```
public class Test {
    /**
     * 基于通归求解
     *
     * @param number 色子个数
     * @param max    色子的最大值
     */
```

```
    public static void printProbability(int number, int max) {
        if (number < 1 || max < 1) {
            return;
        }

        int maxSum = number * max;
        int[] probabilities = new int[maxSum - number + 1];
        probability(number, probabilities, max);

        double total = 1;
        for (int i = 0; i < number; i++) {
            total *= max;
        }

        for (int i = number; i <= maxSum; i++) {
            double ratio = probabilities[i - number] / total;
            System.out.printf("%-8.4f", ratio);
        }

        System.out.println();

    }

    /**
     * @param number        色子个数
     * @param probabilities 不同色子数出现次数的计数数组
     * @param max           色子的最大值
     */
    private static void probability(int number, int[] probabilit
ies, int max) {
        for (int i = 1; i <= max; i++) {
            probability(number, number, i, probabilities, max);
        }
    }

    /**
     * @param original      总的色子数
     * @param current       剩余要处理的色子数
     * @param sum           已经前面的色子数和
     * @param probabilities 不同色子数出现次数的计数数组
```

```
     * @param max           色子的最大值
     */
    private static void probability(int original, int current, i
nt sum, int[] probabilities, int max) {
        if (current == 1) {
            probabilities[sum - original]++;
        } else {
            for (int i = 1; i <= max; i++) {
                probability(original, current - 1, i + sum, prob
abilities, max);
            }
        }
    }

    /**
     * 基于循环求解
     * @param number 色子个数
     * @param max    色子的最大值
     */
    public static void printProbability2(int number, int max) {
        if (number < 1 || max < 1) {
            return;
        }

        int[][] probabilities = new int[2][max * number + 1];
        // 数据初始化
        for (int i = 0; i < max * number + 1; i++) {
            probabilities[0][i] = 0;
            probabilities[1][i] = 0;
        }

        // 标记当前要使用的是第0个数组还是第1个数组
        int flag = 0;

        // 抛出一个骰子时出现的各种情况
        for (int i = 1; i <= max; i++) {
            probabilities[flag][i] = 1;
        }

        // 抛出其它骰子
```

```
        for (int k = 2; k <= number; k++) {
            // 如果抛出了k个骰子，那么和为[0, k-1]的出现次数为0
            for (int i = 0; i < k; i++) {
                probabilities[1 - flag][i] = 0;
            }

            // 抛出k个骰子，所有和的可能
            for (int i = k; i <= max * k; i++) {
                probabilities[1 - flag][i] = 0;

                // 每个骰子的出现的所有可能的点数
                for (int j = 1; j <= i && j <= max; j++) {
                    // 统计出和为i的点数出现的次数
                    probabilities[1 - flag][i] += probabilities[
flag][i - j];
                }
            }

            flag = 1 - flag;
        }


        double total = 1;
        for (int i = 0; i < number; i++) {
            total *= max;
        }

        int maxSum = number * max;
        for (int i = number; i <= maxSum; i++) {
            double ratio = probabilities[flag][i] / total;
            System.out.printf("%-8.4f", ratio);
        }

        System.out.println();
    }
}
```

一、题目

从扑克牌中随机抽5张牌，判断是不是一个顺子， 即这5张牌是不是连续的。2～10为数字本身， A为1。 J为11、Q为12、 为13。小王可以看成任意数字。

二、解题思路

我们可以把5张牌看成由5个数字组成的数组。大、小王是特殊的数字，我们不妨把它们都定义为0，这样就能和其他扑克牌区分开来了。

接下来我们分析怎样判断5个数字是不是连续的，最直观的方法是把数组排序。值得注意的是，由于0可以当成任意数字，我们可以用0去补满数组中的空缺。如果排序之后的数组不是连续的，即相邻的两个数字相隔若干个数字，但只要我们有足够的。可以补满这两个数字的空缺，这个数组实际上还是连续的。举个例子，数组排序之后为{0，1，3，4，5}在1和3之间空缺了一个2，刚好我们有一个0，也就是我们可以把它当成2去填补这个空缺。

于是我们需要做3 件事情： 首先把数组排序，再统计数组中**0** 的个数，最后统计排序之后的数组中相邻数字之间的空缺总数。如果空缺的总数小于或者等于**0** 的个数，那么这个数组就是连续的：反之则不连续。

最后，我们还需要注意一点： 如果数组中的非0 数字重复出现，则该数组不是连续的。换成扑克牌的描述方式就是如果一副牌里含有对子，则不可能是顺子。

三、解题代码

```
public class Test {
    /**
     * 题目：从扑克牌中随机抽5张牌，判断是不是一个顺子， 即这5张牌是不是连续的。
     * 2～10为数字本身， A为1。 J为11、Q为12、 为13。小王可以看成任意数字。
     * @param numbers
     * @return
     */
    public static boolean isContinuous(int[] numbers) {
```

```
        if (numbers == null || numbers.length != 5) {
            return false;
        }

        // 对元素进行排序
        Arrays.sort(numbers);
        int numberOfZero = 0;
        int numberOfGap = 0;
        for (int i = 0; i < numbers.length && numbers[i] == 0; i
++) {
            numberOfZero++;
        }

        // 一副牌中不可能有两个以上的王
        if(numberOfZero > 2) {
            return false;
        }

        // 第一个非0元素的位置
        int small = numberOfZero;
        int big = small + 1;

        while (big < numbers.length) {
            if (numbers[small] == numbers[big]) {
                return false;
            }

            numberOfGap += (numbers[big] - numbers[small] - 1);
            small = big;
            big++;
        }

        return numberOfGap <= numberOfZero;
    }
}
```

一、题目

0, 1, … , n-1 这n个数字排成一个圈圈，从数字0开始每次从圆圈里删除第m个数字。求出这个圈圈里剩下的最后一个数字。

二、解题思路

创建一个总共有n 个结点的环形链表，然后每次在这个链表中删除第m 个结点。

三、解题代码

```
public static int lastRemaining(int n, int m) {
    if (n < 1 || m < 1) {
        return -1;
    }

    List<Integer> list = new LinkedList<>();
    for (int i = 0; i < n; i++) {
        list.add(i);
    }

    // 要删除元素的位置
    int idx = 0;

    while (list.size() > 1) {

        // 只要移动m-1次就可以移动到下一个要删除的元素上
        for (int i = 1; i < m; i++) {
            idx = (idx + 1) % list.size();
        }

        list.remove(idx);
    }

    return list.get(0);
}
```

一、题目

写一个函数，求两个整数之和，要求在函数体内不得使用＋、－、×、÷四则运算符号。

二、解题思路

5 的二进制是101, 17 的二进制是10001 。

试着把计算分成三步：

第一步各位相加但不计进位， 得到的结果是10100 （ 最后一位两个数都是1,相加的结果是二进制的10 ）。这一步不计进位， 因此结果仍然是0 。

第二步记下进位。在这个例子中只在最后一位相加时产生一个进位，结果是二进制的10 。

第三步把前两步的结果相加，得到的结果是10110 ， 转换成十进制正好是22。由此可见三步走的策略对二进制也是适用的。

接下来我们试着把二进制的加法用位运算来替代。

第一步不考虑进位对每一位相加。0加0 、1加1的结果都0。 0加1 、1 加0的结果都是1 。我们注意到，这和异或的结果是一样的。对异或而言， 0和0、1和1异或的结果是0， 而0和1 、1和0的异或结果是1 。

接着考虑第二步进位，对加0 、0 加1 、1加0而言， 都不会产生进位，只有1加1时，会向前产生一个进位。此时我们可以想象成是两个数先做位与运算，然后再向左移动一位。只有两个数都是1的时候，位与得到的结果是1，其余都是0。

第三步把前两个步骤的结果相加。第三步相加的过程依然是重复前面两步， 直到不产生进位为止。

三、解题代码

```
public class Test {
    public static int add(int x, int y) {
        int sum;
        int carry;

        do {
            sum = x ^ y;
            // x&y的某一位是1说明，它是它的前一位的进位，所以向左移动一位
            carry = (x & y) << 1;

            x = sum;
            y = carry;
        } while (y != 0);

        return x;
    }
}
```

一、题目

实现一个函数stringToInt,实现把字符串转换成整数这个功能，不能使用atoi或者其他类似的库函数。

二、解题代码

这看起来是很简单的题目，实现基本功能 ，大部分人都能用10行之内的代码解决。可是，当我们要把很多特殊情况即测试用例都考虑进去，却不是件容易的事。解决数值转换问题本身并不难，但我希望在写转换数值的代码之前，应聘者至少能把空指针，空字符串""，正负号，溢出等方方面面的测试用例都考虑到，并且在写代码的时候对这些特殊的输入都定义好合理的输出。当然，这些输出并不一定要和atoi完全保持一致，但必须要有显式的说明，和面试官沟通好。

这个应聘者最大的问题就是还没有养成在写代码之前考虑所有可能的测试用例的习惯，逻辑不够严谨，因此一开始的代码只处理了最基本的数值转换。后来我每次提醒他一处特殊的测试用例之后，他改一处代码。尽管他已经做了两次修改，但仍然有不少很明显的漏洞，特殊输入空字符串""，边界条件比如最大的正整数与最小的负整数等。由于这道题思路本身不难，因此我希望他把问题考虑得极可能周到，代码尽量写完整。

三、解题思路

```
public class Test {

    /**
     * 题目：实现一个函数stringToInt,实现把字符串转换成整数这个功能，
     * 不能使用atoi或者其他类似的库函数。
     *
     * @param num
     * @return
     */
    public static int stringToInt(String num) {

        if (num == null || num.length() < 1) {
```

```
                throw new NumberFormatException(num);
            }

            char first = num.charAt(0);
            if (first == '-') {
                return parseString(num, 1, false);
            } else if (first == '+') {
                return parseString(num, 1, true);
            } else if (first <= '9' && first >= '0') {
                return parseString(num, 0, true);
            } else {
                throw new NumberFormatException(num);
            }
        }

        /**
         * 判断字符是否是数字
         *
         * @param c 字符
         * @return true是，false否
         */
        private static boolean isDigit(char c) {
            return c >= '0' && c <= '9';
        }

        /**
         * 对字符串进行解析
         *
         * @param num      数字串
         * @param index    开始解析的索引
         * @param positive 是正数还是负数
         * @return 返回结果
         */
        private static int parseString(String num, int index, boolean
    positive) {

            if (index >= num.length()) {
                throw new NumberFormatException(num);
            }
```

```
        int result;
        long tmp = 0;
        while (index < num.length() && isDigit(num.charAt(index)
)) {
            tmp = tmp * 10 + num.charAt(index) - '0';
            // 保证求的得的值不超出整数的最大绝对值
            if (tmp > 0x8000_0000L) {
                throw new NumberFormatException(num);
            }
            index++;
        }

        if (positive) {
            if (tmp >= 0x8000_0000L) {
                throw new NumberFormatException(num);
            } else {
                result = (int) tmp;
            }
        } else {
            if (tmp == 0x8000_0000L) {
                result = 0x8000_0000;
            } else {
                result = (int) -tmp;
            }
        }

        return result;
    }
}
```

一、题目

求树中两个结点的最低公共祖先，此树不是二叉树，并且没有指向父节点的指针。

二、解题思路

我们首先得到一条从根结点到树中某一结点的路径，这就要求在遍历的时候，有一个辅助内存来保存路径．

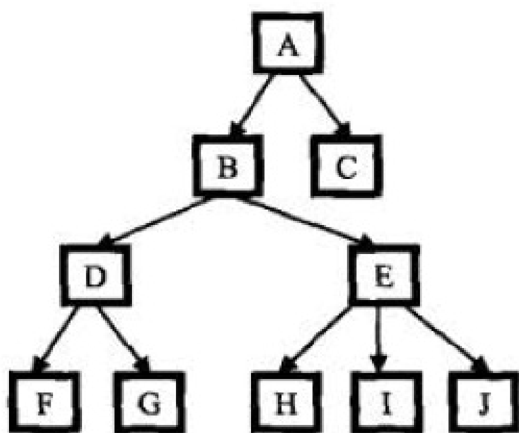


图 7.2 一棵普通的树，树中的结点没有指向父结点的指针

比如我们用前序遍历的方法来得到从根结点到H 的路径的过程是这样的：（ 1 ）遍历到A，把A 存放到路径中去，路径中只有一个结点A; (2 ）遍历到B，把B 存到路径中去，此时路径为A->B; (3 ）遍历到D，把D 存放到路径中去，此，时路径为A->B->D; (4 ） 遍历到F，把F 存放到路径中去，此时路径为A->B->D->F;(5) F 已经没有子结点了，因此这条路径不可能到这结点H. 把F 从路径中删除，变成A->B->D; (6 ）遍历G. 和结点F 一样，这条路径也不能到达H. 边历完G 之后，路径仍然是A->B->D; (7 ）由于D 的所有子结点都遍历过了，不可能到这结点H，因此D 不在从A 到H 的路径中，把D 从路径中删除，变成A->B; (8 ）遥历E，把E 加入到路径中，此时路径变成A->B->E, (9 ）遍历H，已经到达目标给点， A->B->E 就是从根结点开始到达H 必须经过的路径。

同样，我们也可以得到从根结点开始到达F 必须经过的路径是A->B。接着，我们求出这两个路径的最后公共结点，也就是B. B这个结点也是F 和H 的最低公共祖先．

为了得到从根结点开始到输入的两个结点的两条路径，需要遍历两次树，每边历一次的时间复杂度是O(n)。

三、解题代码

```
public class Test{
    /**
     * 树的结点定义
     */
    private static class TreeNode {
        int val;

        List<TreeNode> children = new LinkedList<>();


        public TreeNode() {
        }

        public TreeNode(int val) {
            this.val = val;
        }

        @Override
        public String toString() {
            return val + "";
        }
    }

    /**
     * 找结点的路径
     *
     * @param root   根结点
     * @param target 目标结点
     * @param path   从根结点到目标结点的路径
     */
    public static void getNodePath(TreeNode root, TreeNode targe
t, List<TreeNode> path) {
        if (root == null) {
            return;
        }

        // 添加当前结点
```

```
        path.add(root);

        List<TreeNode> children = root.children;
        // 处理子结点
        for (TreeNode node : children) {

            if (node == target) {
                path.add(node);
                return;
            } else {
                getNodePath(node, target, path);
            }
        }

        // 现场还原
        path.remove(path.size() - 1);
    }

    /**
     * 找两个路径中的最后一个共同的结点
     *
     * @param p1 路径1
     * @param p2 路径2
     * @return 共同的结点，没有返回null
     */
    public static TreeNode getLastCommonNode(List<TreeNode> p1,
List<TreeNode> p2) {
        Iterator<TreeNode> ite1 = p1.iterator();
        Iterator<TreeNode> ite2 = p2.iterator();
        TreeNode last = null;

        while (ite1.hasNext() && ite2.hasNext()) {
            TreeNode tmp = ite1.next();
            if (tmp == ite2.next()) {
                last = tmp;
            }
        }

        return last;
```

```
    }

    /**
     * 找树中两个结点的最低公共祖先
     * @param root 树的根结点
     * @param p1 结点1
     * @param p2 结点2
     * @return 公共结点，没有返回null
     */
    public static TreeNode getLastCommonParent(TreeNode root, Tr
eeNode p1, TreeNode p2) {
        if (root == null || p1 == null || p2 == null) {
            return null;
        }
        List<TreeNode> path1 = new LinkedList<>();
        getNodePath(root, p1, path1);
        List<TreeNode> path2 = new LinkedList<>();
        getNodePath(root, p2, path2);

        return getLastCommonNode(path1, path2);
    }
}
```

一、题目

在一个长度为n的数组里的所有数字都在0到n-1的范围内。数组中某些数字是重复的，但不知道有几个数字重复了，也不知道每个数字重复了几次。请找出数组中任意一个重复的数字。

举例说明

例如，如果输入长度为7的数组{2,3,1,0,2,5,3}，那么对应的输出是重复的数字2或者3。

二、解题思路

解决这个问题的一个简单的方法是先把输入的数组排序。从排序的数组中找出重复的数字时间很容易的事情，只需要从头到尾扫描排序后的数组就可以了。排序一个长度为n的数组需要O(nlogn)的时间。

还可以利用哈希表来解决这个问题。从头到尾按顺序扫描数组的每个数，每扫描一个数字的时候，都可以用O(1)的时间来判断哈希表里是否已经包含了该数字。如果哈希表里还没有这个数字，就把它加入到哈希表里。如果哈希表里已经存在该数字了，那么就找到一个重复的数字。这个算法的时间复杂度是O(n)，但它提高时间效率是以一个大小为O(n)的哈希表为代价的。我们再看看有没有空间复杂度为O(1)的算法。

我们注意到数组中的数字都在0到n-1中。如果这个数组中没有重复的数字，那么当数组排序之后数字i将出现在下标为i的位置。由于数组中有重复的数字，有些位置可能存在多个数字，同时有些位置可能没有数字。 现在让我们重排这个数组，依然从头到尾一次扫描这个数组中的每个数字。当扫描到下标为i的数字时，首先比较这个数字（用m表示）是不是等于i。如果是，接着扫描下一个数字。如果不是，再拿它和第m个数字进行比较。

如果它和第m个数字相等，就找到了一个重复的数字（该数字在下标为i和m的位置都出现了）。如果它和第m个数字不相等，就把第i个数字和第m个数字交换，把m放到属于它的位置。接下来再重读这个比较、交换的过程，直到我们发现一个重复的数字。

以数组{2,3,1,0,2,5,3}为例来分析找到重复数字的步骤。数组的第0个数字（从0开始计数，和数组的下标保持一致）是2，与它的下标不相等，于是把它和下标为2的数字1交换。交换之后的数组是{1,3,2,0,2,5,3}。此时第0个数字是1，仍然与它的下标不相等，继续把它和下标为1的数字3交换，得到数组{3,1,2,0,2,5,3}.接下来继续交换第0个数字3和第3个数字0，得到数组{0,1,2,3,2,5,3}。此时第0个数字的数值为0，接着扫描下一个数字。在接下来的几个数字中，下标为1,2,3的三个数字分别为1,2,3，它们的下标和数值都分别相等，因此不需要做任何操作。接下来扫描到下标为4的数字2.由于它的数值与它的下标不相等，再比较它和下标为2的数字。注意到此时数组中下标为2的数字也是2，也就是数字在下标为2和下标为4的两个位置都出现了，因此找到一个重复的数字。

三、解题代码

```
public class Test{
    /**
     * 题目：在一个长度为n的数组里的所有数字都在0到n-1的范围内。数组中某些数字是重复的，
     * 但不知道有几个数字重复了，也不知道每个数字重复了几次。请找出数组中任意一个重复的数字。
     * 例如，如果输入长度为7的数组{2,3,1,0,2,5,3}，那么对应的输出是重复的数字2或者。
     *
     * @param number
     * @return
     */
    public static int duplicate(int[] number) {
        if (number == null || number.length < 1) {
            return -1;
        }

        // 判断输入的是否在[0, number.length-1]之间
        for (int i : number) {
            if (i < 0 || i >= number.length) {
                return -1;
            }
        }

        for (int i = 0; i < number.length; i++) {
```

```
                // 当number[i]与i不相同的时候一直交换
                while (number[i] != i) {
                    // 如果i位置与number[i]位置的数字相同，说明有重复数字
                    if (number[i] == number[number[i]]) {
                        return number[i];
                    }
                    // 如果不同就交换
                    else {
                        swap(number, i, number[i]);
                    }
                }
            }
            return -1;
        }

        private static void swap(int[] data, int x, int y) {
            int tmp = data[x];
            data[x] = data[y];
            data[y] = tmp;
        }
    }
```

一、题目

给定一个数组A[0,1,...,n-1],请构建一个数组B[0,1,...,n-1],其中B中的元素B[i]=A[0]×A[1]×...×A[i-1]×A[i+1]×...×A[n-1],不能使用除法。

二、解题思路

B[i]的值可以看作下图的矩阵中每行的乘积。

下三角用连乘可以很容求得，上三角，从下向上也是连乘。

因此我们的思路就很清晰了，先算下三角中的连乘，即我们先算出B[i]中的一部分，然后倒过来按上三角中的分布规律，把另一部分也乘进去。

B_0	1	A_1	A_2	---	A_{n-2}	A_{n-1}
B_1	A_0	1	A_2	---	A_{n-2}	A_{n-1}
B_2	A_0	A_1	1	---	A_{n-2}	A_{n-1}
---	A_0	A_1	---	1	A_{n-2}	A_{n-1}
B_{n-2}	A_0	A_1	---	A_{n-3}	1	A_{n-1}
B_{n-1}	A_0	A_1	---	A_{n-3}	A_{n-2}	1

三、解题代码

```
public class Test{
    public static double[] multiply(double[] data) {
        if (data == null || data.length < 2) {
            return null;
        }

        double[] result = new double[data.length];

        // result[0]取1
        result[0] = 1;
        for (int i = 1; i < data.length; i++) {
            // 第一步每个result[i]都等于于data[0]*data[1]...data[i-
1]
            // 当i=n-1时，此时result[n-1]的结果已经计算出来了
            result[i] = result[i -1] * data[i - 1];
        }

        // tmp保存data[n-1]*data[n-2]...data[i+1]的结果
        double tmp = 1;
        // 第二步求data[n-1]*data[n-2]...data[i+1]
        // result[n-1]的结果已经计算出来，所以从data.length-2开始操作
        for (int i = data.length - 2; i >= 0; i--) {
            tmp *= data[i + 1];
            result[i] *= tmp;
        }

        return result;
    }
}
```

一、题目

请实现一个函数用来匹配包含 . 和 的正则表达式。模式中的字符'.'表示任意一个字符，而 表示它前面的字符可以出现任意次（含0次）。本题中，匹配是指字符串的所有字符匹配整个模式。例如，字符串“aaa”与模式“a.a”和“abaca”匹配，但与“aa.a”及“ab*a”均不匹配。

二、解题思路

假设字符串为str，模式串为pattern，考虑以下情况：

A. 模式串下一个字符为 * ：

如果当前字符匹配，三种可能：

1、模式串当前字符出现0次，即 * 表示当前字符出现0次，则str[i]->str[i],pattern[j]->pattern[j+2];

2、模式串当前字符出现1次，即 * 表示当前字符出现1次，则str[i]->str[i+1],pattern[j]->pattern[j+2];

3、模式串当前字符出现2次或2次以上，即 * 表示当前字符出现2次或以上，则str[i]->str[i+1],pattern[j]->pattern[i];

如果当前字符不匹配，则只能让 * 表示当前字符出现0次，则str[i]->str[i]，pattern[j]->pattern[j+2];

B. 模式串下一个字符不为 *

如果当前字符匹配，则str=str+1,pattern=pattern+1.

三、解题代码

```
public class Test {

    /**
     * 题目：请实现一个函数用来匹配包含'.'和'*'的正则表达式。模式中的字符'
.'表示任意一个字符，
```

```
     * 而'*'表示它前面的字符可以出现任意次（含0次）。本题中，匹配是指字符串
的所有字符匹配整个模式。
     *
     * @param input
     * @param pattern
     * @return
     */
    public static boolean match(String input, String pattern) {
        if (input == null || pattern == null) {
            return false;
        }

        return matchCore(input, 0, pattern, 0);
    }

    private static boolean matchCore(String input, int i, String
 pattern, int p) {

        // 匹配串和模式串都到达尾，说明成功匹配
        if (i >= input.length() && p >= pattern.length()) {
            return true;
        }

        // 只有模式串到达结尾，说明匹配失败
        if (i != input.length() && p >= pattern.length()) {
            return false;
        }

        // 模式串未结束，匹配串有可能结束有可能未结束

        // p位置的下一个字符中为*号
        if (p + 1 < pattern.length() && pattern.charAt(p + 1) ==
'*') {

            // 匹配串已经结束
            if (i >= input.length()) {
                return matchCore(input, i, pattern, p + 2);
            }
            // 匹配串还没有结束
            else {
```

```
                if (pattern.charAt(p) == input.charAt(i) || patt
ern.charAt(p) == '.') {
                    return
                            // 匹配串向后移动一个位置，模式串向后移动两
个位置
                            matchCore(input, i + 1, pattern, p +
2)
                                    // 匹配串向后移动一个位置，模式串
不移动
                                    || matchCore(input, i + 1, p
attern, p)
                                    // 匹配串不移动，模式串向后移动两
个位置
                                    || matchCore(input, i, patte
rn, p + 2);
                } else {
                    return matchCore(input, i, pattern, p + 2);
                }
            }
        }

        // 匹配串已经结束
        if (i >= input.length()) {
            return false;
        }
        // 匹配串还没有结束
        else {
            if (input.charAt(i) == pattern.charAt(p) || pattern.
charAt(p) == '.') {
                return matchCore(input, i + 1, pattern, p + 1);
            }
        }


        return false;
    }
}
```

时间： 2018-05-12 01:44:07

一、题目

请实现一个函数用来判断字符串是否表示数值（包括整数和小数）。

例子说明

例如，字符串“+100”，“5e2”，“-123”，“3.1416”及”-1E-16”都表示数值，但“12e”,”1a3.14”,”1.2.3”,”+-5”及“12e+5.4”都不是。

二、解题思路

在数值之前可能有一个表示正负的’-‘或者’+’。接下来是若干个0到9的数位表示数值的整数部分（在某些小数里可能没有数值的整数部分）。如果数值是一个小数，那么在小数点后面可能会有若干个0到9的数位表示数值的小数部分。如果数值用科学计数法表示，接下来是一个’e’或者‘E’，以及紧跟着的一个整数（可以有正负号）表示指数。

判断一个字符串是否符合上述模式时，首先看第一个字符是不是正负号。如果是，在字符串上移动一个字符，继续扫描剩余的字符串中0到9的数位。如果是一个小数，则将遇到小数点。另外，如果是用科学计数法表示的数值，在整数或者小数的后面还有可能遇到’e’或者’E’。

三、解题代码

```
public class Test {
    /**
     * 题目：请实现一个函数用来判断字符串是否表示数值（包括整数和小数）。
     *
     * @param string
     * @return
     */
    public static boolean isNumeric(String string) {
        if (string == null || string.length() < 1) {
            return false;
        }
        int index = 0;
```

```
        if (string.charAt(index) == '+' || string.charAt(index)
== '-') {
            index++;
        }

        // 已经到达字符串的末尾了
        if (index >= string.length()) {
            return false;
        }

        boolean numeric = true;
        index = scanDigits(string, index);
        // 还未到字符串的末尾
        if (index < string.length()) {
            // 如果是小数点
            if (string.charAt(index) == '.') {
                // 移动到下一个位置
                index++;
                index = scanDigits(string, index);

                // 已经到了字符串的末尾了
                if (index >= string.length()) {
                    numeric = true;
                }
                // 还未到字符串结束位置
                else if (index < string.length() && (string.char
At(index) == 'e' || string.charAt(index) == 'E')) {
                    numeric = isExponential(string, index);
                } else {
                    numeric = false;
                }
            }
            // 如果是指数标识
            else if (string.charAt(index) == 'e' || string.charA
t(index) == 'E') {
                numeric = isExponential(string, index);
            } else {
                numeric = false;
            }
```

```
            return numeric;
        }
        // 已经到了字符串的末尾了，说明其没有指数部分
        else {
            return true;
        }

    }

    /**
     * 判断是否是科学计数法的结尾部分，如E5，e5，E+5，e-5，e(E)后面接整数
     *
     * @param string 字符串
     * @param index  开始匹配的位置
     * @return 匹配的结果
     */
    private static boolean isExponential(String string, int inde
x) {

        if (index >= string.length() || (string.charAt(index) !=
'e' && string.charAt(index) != 'E')) {
            return false;
        }

        // 移动到下一个要处理的位置
        index++;

        // 到达字符串的末尾，就返回false
        if (index >= string.length()) {
            return false;
        }

        if (string.charAt(index) == '+' || string.charAt(index)
== '-') {
            index++;
        }

        // 到达字符串的末尾，就返回false
        if (index >= string.length()) {
```

```
            return false;
        }

        index = scanDigits(string, index);

        // 如果已经处理到了的数字的末尾就认为是正确的指数
        return index >= string.length();
    }

    /**
     * 扫描字符串部分的数字部分
     *
     * @param string 字符串
     * @param index  开始扫描的位置
     * @return 从扫描位置开始第一个数字字符的位置
     */
    private static int scanDigits(String string, int index) {
        while (index < string.length() && string.charAt(index) >
= '0' && string.charAt(index) <= '9') {
            index++;
        }
        return index;
    }
}
```

一、题目

请实现一个函数用来找出字符流中第一个只出现一次的字符。

举例说明

例如，当从字符流中只读出前两个字符“Go”时，第一个只出现一次的字符是‘g’。当从该字符流中读出前六个字符“google”时，第一个只出现1次的字符是”l”。

二、解题思路

字符只能一个接着一个从字符流中读出来。可以定义一个数据容器来保存字符在字符流中的位置。当一个字符第一次从字符流中读出来时，把它在字符流中的位置保存到数据容器里。当这个字符再次从字符流中被读出来时，那么它就不是只出现一次的字符，也就可以被忽略了。这时把它在数据容器里保存的值更新成一个特殊的值（比如负值）。

为了尽可能高校地解决这个问题，需要在O(1)时间内往容器里插入一个字符，以及更新一个字符对应的值。这个容器可以用哈希表来实现。用字符的ASCII码作为哈希表的键值，而把字符对应的位置作为哈希表的值。

三、解题代码

```
public class Test {
    /**
     * 题目：请实现一个函数用来找出字符流中第一个只出现一次的字符。
     */
    private static class CharStatistics {
        // 出现一次的标识
        private int index = 0;
        private int[] occurrence = new int[256];

        public CharStatistics() {
            for (int i = 0; i < occurrence.length; i++) {
                occurrence[i] = -1;
            }
```

```
        }

        private void insert(char ch) {
            if (ch > 255) {
                throw new IllegalArgumentException( ch + "must b
e a ASCII char");
            }

            // 只出现一次
            if (occurrence[ch] == -1) {
                occurrence[ch] = index;
            } else {
                // 出现了两次
                occurrence[ch] = -2;
            }

            index++;
        }

        public char firstAppearingOnce(String data) {
            if (data == null) {
                throw new IllegalArgumentException(data);
            }

            for (int i = 0; i < data.length(); i++) {
                insert(data.charAt(i));
            }
            char ch = '\0';
            // 用于记录最小的索引，对应的就是第一个不重复的数字
            int minIndex = Integer.MAX_VALUE;
            for (int i = 0; i < occurrence.length; i++) {
                if (occurrence[i] >= 0 && occurrence[i] < minInd
ex) {
                    ch = (char) i;
                    minIndex = occurrence[i];
                }
            }

            return ch;
        }
```

```
    }
}
```

一、题目

一个链表中包含环，如何找出环的入口结点？

二、解题思路

可以用两个指针来解决这个问题。先定义两个指针P1和P2指向链表的头结点。如果链表中环有n个结点，指针P1在链表上向前移动n步，然后两个指针以相同的速度向前移动。当第二个指针指向环的入口结点时，第一个指针已经围绕着环走了一圈又回到了入口结点。

剩下的问题就是如何得到环中结点的数目。我们在面试题15的第二个相关题目时用到了一快一慢的两个指针。如果两个指针相遇，表明链表中存在环。两个指针相遇的结点一定是在环中。可以从这个结点出发，一边继续向前移动一边计数，当再次回到这个结点时就可以得到环中结点数了。

三、解题代码

```
public class Test {
    private static class ListNode {
        private int val;
        private ListNode next;

        public ListNode() {
        }

        public ListNode(int val) {
            this.val = val;
        }

        @Override
        public String toString() {
            return val +"";
        }
    }
    public static ListNode meetingNode(ListNode head) {
```

```
        ListNode fast = head;
        ListNode slow = head;

        while (fast != null && fast.next != null) {
            fast = fast.next.next;
            slow = slow.next;
            if (fast == slow) {
                break;
            }
        }

        // 链表中没有环
        if (fast == null || fast.next == null) {
            return null;
        }
        return fast;
    }
    public ListNode EntryNodeOfLoop(ListNode pHead) {
        ListNode meetingNode=meetingNode(pHead);
        if(meetingNode==null)
            return null;
//        得到环中的节点个数
        int nodesInLoop=1;
        ListNode p1=meetingNode;
        while(p1.next!=meetingNode){
            p1=p1.next;
            ++nodesInLoop;
        }
//        移动p1
        p1=pHead;
        for(int i=0;i<nodesInLoop;i++){
            p1=p1.next;
        }
//        移动p1，p2
        ListNode p2=pHead;
        while(p1!=p2){
            p1=p1.next;
            p2=p2.next;
        }
        return p1;
```

```
    }
}
```

一、题目

在一个排序的链表中，如何删除重复的结点？

例如，链表1->2->3->3->4->4->5 处理后为 1->2->5

二、解题思路

解决这个问题的第一步是确定删除的参数。当然这个函数需要输入待删除链表的头结点。头结点可能与后面的结点重复，也就是说头结点也可能被删除，所以在链表头添加一个结点。

接下来我们从头遍历整个链表。如果当前结点的值与下一个结点的值相同，那么它们就是重复的结点，都可以被删除。为了保证删除之后的链表仍然是相连的而没有中间断开，我们要把当前的前一个结点和后面值比当前结点的值要大的结点相连。我们要确保prev要始终与下一个没有重复的结点连接在一起。

三、解题代码

```
public static ListNode deleteDuplication(ListNode pHead) {

        ListNode first = new ListNode(-1);//设置一个trick

        first.next = pHead;

        ListNode p = pHead;
        ListNode last = first;
        while (p != null && p.next != null) {
            if (p.val == p.next.val) {
                int val = p.val;
                while (p!= null&&p.val == val)
                    p = p.next;
                last.next = p;
            } else {
                last = p;
                p = p.next;
            }
        }
        return first.next;
    }
```

一、题目

给定一棵二叉树和其中的一个结点，如何找出中序遍历顺序的下一个结点？树中的结点除了有两个分别指向左右子结点的指针以外，还有一个指向父节点的指针。

二、解题思路

如果一个结点有右子树，那么它的下一个结点就是它的右子树中的左子结点。也就是说右子结点出发一直沿着指向左子结点的指针，我们就能找到它的下一个结点。

接着我们分析一个结点没有右子树的情形。如果结点是它父节点的左子结点，那么它的下一个结点就是它的父结点。

如果一个结点既没有右子树，并且它还是它父结点的右子结点，这种情形就比较复杂。我们可以沿着指向父节点的指针一直向上遍历，直到找到一个是它父结点的左子结点的结点。如果这样的结点存在，那么这个结点的父结点就是我们要找的下一个结点。

三、解题代码

```
public class Test {
    private static class BinaryTreeNode {
        private int val;
        private BinaryTreeNode left;
        private BinaryTreeNode right;
        private BinaryTreeNode parent;

        public BinaryTreeNode() {
        }

        public BinaryTreeNode(int val) {
            this.val = val;
        }

        @Override
        public String toString() {
```

```
            return val + "";
        }
    }

    public static BinaryTreeNode getNext(BinaryTreeNode node) {
        if (node == null) {
            return null;
        }

        // 保存要查找的下一个节点
        BinaryTreeNode target = null;

        if (node.right != null) {
            target = node.right;
            while (target.left != null) {
                target = target.left;
            }

            return target;
        } else if (node.parent != null){
            target = node.parent;
            BinaryTreeNode cur = node;
            // 如果父新结点不为空，并且，子结点不是父结点的左孩子
            while (target != null && target.left != cur) {
                cur = target;
                target = target.parent;

            }

            return target;
        }

        return null;
    }
}
```

一、题目

请实现一个函数来判断一棵二叉树是不是对称的。如果一棵二叉树和它的镜像一样，那么它是对称的。

二、解题思路

通常我们有三种不同的二叉树遍历算法，即前序遍历、中序遍历和后序遍历。在这三种遍历算法中，都是先遍历左子结点再遍历右子结点。我们是否可以定义一种遍历算法，先遍历右子结点再遍历左子结点？比如我们针对前序遍历定义一种对称的遍历算法，即先遍历父节点，再遍历它的右子结点，最后遍历它的左子结点。

我们发现可以用过比较二叉树的前序遍历序列和对称前序遍历序列来判断二叉树是不是对称的。如果两个序列一样，那么二叉树就是对称的。

三、解题代码

```
public class Test {
    private static class BinaryTreeNode {
        private int val;
        private BinaryTreeNode left;
        private BinaryTreeNode right;

        public BinaryTreeNode() {
        }

        public BinaryTreeNode(int val) {
            this.val = val;
        }

        @Override
        public String toString() {
            return val + "";
        }
    }
```

```
    public static boolean isSymmetrical(BinaryTreeNode root) {
        return isSymmetrical(root, root);
    }

    private static boolean isSymmetrical(BinaryTreeNode left, Bi
naryTreeNode right) {

        if (left == null && right == null) {
            return true;
        }

        if (left == null || right == null) {
            return false;
        }

        if (left.val != right.val ) {
            return false;
        }

        return isSymmetrical(left.left, right.right) && isSymmet
rical(left.right, right.left);
    }
}
```

一、题目

从上到下按层打印二叉树，同一层的结点按从左到右的顺序打印，每一层打印一行。

二、解题思路

用一个队列来保存将要打印的结点。为了把二叉树的每一行单独打印到一行里，我们需要两个变量：一个变量表示在当前的层中还没有打印的结点数，另一个变量表示下一层结点的数目。

三、解题代码

```
public class Test {
    private static class BinaryTreeNode {
        private int val;
        private BinaryTreeNode left;
        private BinaryTreeNode right;

        public BinaryTreeNode() {
        }

        public BinaryTreeNode(int val) {
            this.val = val;
        }

        @Override
        public String toString() {
            return val + "";
        }
    }

    /**
     * 题目：从上到下按层打印二叉树，同一层的结点按从左到右的顺序打印，每一层打印一行。
     * @param root
```

```
     */
    public static void print(BinaryTreeNode root) {
        if (root == null) {
            return;
        }

        List<BinaryTreeNode> list = new LinkedList<>();
        BinaryTreeNode node;
        // 当前层的结点个数
        int current = 1;
        // 记录下一层的结点个数
        int next = 0;
        list.add(root);

        while (list.size() > 0) {
            node = list.remove(0);
            current--;
            System.out.printf("%-3d", node.val);

            if (node.left != null) {
                list.add(node.left);
                next++;
            }
            if (node.right != null) {
                list.add(node.right);
                next++;
            }

            if (current ==0) {
                System.out.println();
                current = next;
                next = 0;
            }
        }
    }
}
```

一、题目

请实现一个函数按照之字形顺序打印二叉树，即第一行按照从左到右的顺序打印，第二层按照从右到左的顺序打印，即第一行按照从左到右的顺序打印，第二层按照从右到左顺序打印，第三行再按照从左到右的顺序打印，其他以此类推。

二、解题思路

按之字形顺序打印二叉树需要两个栈。我们在打印某一行结点时，把下一层的子结点保存到相应的栈里。如果当前打印的是奇数层，则先保存左子结点再保存右子结点到一个栈里；如果当前打印的是偶数层，则先保存右子结点再保存左子结点到第二个栈里。

三、解题代码

```
public class Test {
    private static class BinaryTreeNode {
        private int val;
        private BinaryTreeNode left;
        private BinaryTreeNode right;

        public BinaryTreeNode() {
        }

        public BinaryTreeNode(int val) {
            this.val = val;
        }

        @Override
        public String toString() {
            return val + "";
        }
    }

    public static void print(BinaryTreeNode root) {
```

```
        if (root == null) {
            return;
        }


        List<BinaryTreeNode> current = new LinkedList<>();
        List<BinaryTreeNode> reverse = new LinkedList<>();
        int flag = 0;
        BinaryTreeNode node;
        current.add(root);

        while (current.size() > 0) {

            // 从最后一个开始取
            node = current.remove(current.size() - 1);

            System.out.printf("%-3d", node.val);

            // 当前是从左往右打印的，那就按从左往右入栈
            if (flag == 0) {
                if (node.left != null) {
                    reverse.add(node.left);
                }

                if (node.right != null) {
                    reverse.add(node.right);
                }


            }
            // 当前是从右往左打印的，那就按从右往左入栈
            else {
                if (node.right != null) {
                    reverse.add(node.right);
                }

                if (node.left != null) {
                    reverse.add(node.left);
                }
            }
```

```
                if (current.size() == 0) {
                    flag = 1 - flag;
                    List<BinaryTreeNode> tmp = current;
                    current = reverse;
                    reverse = tmp;
                    System.out.println();
                }
            }
        }
}
```

一、题目

给定一棵二叉搜索树，请找出其中的第k大的结点。

二、解题思路

如果按照中序遍历的顺序遍历一棵二叉搜索树，遍历序列的数值是递增排序的。只需要用中序遍历算法遍历一棵二叉搜索树，就很容易找出它的第k大结点。

三、解题代码

```
public class Test {
    private static class BinaryTreeNode {
        private int val;
        private BinaryTreeNode left;
        private BinaryTreeNode right;

        public BinaryTreeNode() {
        }

        public BinaryTreeNode(int val) {
            this.val = val;
        }

        @Override
        public String toString() {
            return val + "";
        }
    }

    public static BinaryTreeNode kthNode(BinaryTreeNode root, int
 k) {
        if (root == null || k < 1) {
            return null;
        }
```

```
        int[] tmp = {k};
        return kthNodeCore(root, tmp);
    }

    private static BinaryTreeNode kthNodeCore(BinaryTreeNode roo
t, int[] k) {
        BinaryTreeNode result = null;

        // 先成左子树中找
        if (root.left != null) {
          result =  kthNodeCore(root.left, k);
        }

        // 如果在左子树中没有找到
        if (result == null) {
            // 说明当前的根结点是所要找的结点
            if(k[0] == 1) {
                result = root;
            } else {
                // 当前的根结点不是要找的结点，但是已经找过了，所以计数器
减一
                k[0]--;
            }
        }

        // 根结点以及根结点的左子树都没有找到，则找其右子树
        if (result == null && root.right != null) {
            result = kthNodeCore(root.right, k);
        }

        return result;
    }
}
```

一、题目

如何得到一个数据流中的中位数？如果从数据流中读出奇数个数值，那么中位数就是所有值排序之后位于中间的数值。如果数据流中读出偶数个数值，那么中位数就是所有数值排序之后中间两个数的平均值。

二、解题思路

如果能够保证数据容器左边的数据都小于右边的数据，这样即使左、右两边内部的数据没有排序，也可以根据左边最大的数及右边最小的数得到中位数。如何快速从一个容器中找出最大数？用最大堆实现这个数据容器，因为位于堆顶的就是最大的数据。同样，也可以快速从最小堆中找出最小数。　　因此可以用如下思路来解决这个问题：用一个最大堆实现左边的数据容器，用最小堆实现右边的数据容器。往堆中插入一个数据的时间效率是O(logn).由于只需O(1)时间就可以得到位于堆顶的数据，因此得到中位数的时间效率是O(1).

接下来考虑用最大堆和最小堆实现的一些细节。首先要保证数据平均分配到两个堆中，因此两个堆中数据的数目之差不能超过1（为了实现平均分配，可以在数据的总数目是偶数时把新数据插入到最小堆中，否则插入到最大堆中）。

还要保证最大堆中里的所有数据都要小于最小堆中的数据。当数据的总数目是偶数时，按照前面分配的规则会把新的数据插入到最小堆中。如果此时新的数据比最大堆中的一些数据要小，怎么办呢？

可以先把新的数据插入到最大堆中，接着把最大堆中的最大的数字拿出来插入到最小堆中。由于最终插入到最小堆的数字是原最大堆中最大的数字，这样就保证了最小堆中的所有数字都大于最大堆中的数字。

当需要把一个数据插入到最大堆中，但这个数据小于最小堆里的一些数据时，这个情形和前面类似。

三、解题代码

```
public class Test {
    private static class Heap<T> {
```

```
        // 堆中元素存放的集合
        private List<T> data;
        // 比较器
        private Comparator<T> cmp;

        /**
         * 构造函数
         *
         * @param cmp 比较器对象
         */
        public Heap(Comparator<T> cmp) {
            this.cmp = cmp;
            this.data = new ArrayList<>(64);
        }

        /**
         * 向上调整堆
         *
         * @param idx 被上移元素的起始位置
         */
        public void shiftUp(int idx) {
            // 检查是位置是否正确
            if (idx < 0 || idx >= data.size()) {
                throw new IllegalArgumentException(idx + "");
            }

            // 获取开始调整的元素对象
            T intent = data.get(idx);

            // 如果不是根元素，则需要上移
            while (idx > 0) {
                // 找父元素对象的位置
                int parentIdx = (idx - 1) / 2;
                // 获取父元素对象
                T parent = data.get(parentIdx);
                //上移的条件，子节点比父节点大，此处定义的大是以比较器返
回值为准
                if (cmp.compare(intent, parent) > 0) {
                    // 将父节点向下放
                    data.set(idx, parent);
```

```
                idx = parentIdx;
                // 记录父节点下放的位置
            }
            // 子节点不比父节点大，说明父子路径已经按从大到小排好顺序
了，不需要调整了
            else {
                break;
            }
        }

        // index此时记录是的最后一个被下放的父节点的位置（也可能是自
身），
        // 所以将最开始的调整的元素值放入index位置即可
        data.set(idx, intent);
    }

    /**
     * 向下调整堆
     *
     * @param idx 被下移的元素的起始位置
     */
    public void shiftDown(int idx) {
        // 检查是否位置是否正确
        if (idx < 0 || idx >= data.size()) {
            throw new IllegalArgumentException(idx + "");
        }

        // 获取开始调整的元素对象
        T intent = data.get(idx);
        // 获取开始调整的元素对象的左子结点的元素位置
        int leftIdx = idx * 2 + 1;
        // 如果有左子结点
        while (leftIdx < data.size()) {
            // 取左子结点的元素对象，并且假定其为两个子结点中最大的
            T maxChild = data.get(leftIdx);
            // 两个子节点中最大节点元素的位置，假定开始时为左子结点的
位置
            int maxIdx = leftIdx;

            // 获取右子结点的位置
```

```
                int rightIdx = leftIdx + 1;
                // 如果有右子结点
                if (rightIdx < data.size()) {
                    T rightChild = data.get(rightIdx);
                    // 找出两个子节点中的最大子结点
                    if (cmp.compare(rightChild, maxChild) > 0) {
                        maxChild = rightChild;
                        maxIdx = rightIdx;
                    }
                }

                // 如果最大子节点比父节点大，则需要向下调整
                if (cmp.compare(maxChild, intent) > 0) {
                    // 将较大的子节点向上移
                    data.set(idx, maxChild);
                    // 记录上移节点的位置
                    idx = maxIdx;
                    // 找到上移节点的左子节点的位置
                    leftIdx = 2 * idx + 1;
                }
                // 最大子节点不比父节点大，说明父子路径已经按从大到小排好
顺序了，不需要调整了
                else {
                    break;
                }

            }
            // index此时记录是的最后一个被上移的子节点的位置（也可能是自
身），

            // 所以将最开始的调整的元素值放入index位置即可
            data.set(idx, intent);
        }

        /**
         * 添加一个元素
         *
         * @param item 添加的元素
         */
        public void add(T item) {
            // 将元素添加到最后
```

```
            data.add(item);
            // 上移，以完成重构
            shiftUp(data.size() - 1);
        }

        /**
         * 删除堆顶结点
         *
         * @return 堆顶结点
         */
        public T deleteTop() {
            // 如果堆已经为空，就抛出异常
            if (data.isEmpty()) {
                throw new RuntimeException("The heap is empty.")
;
            }

            // 获取堆顶元素
            T first = data.get(0);
            // 删除最后一个元素
            T last = data.remove(data.size() - 1);

            // 删除元素后，如果堆为空的情况，说明删除的元素也是堆顶元素
            if (data.size() == 0) {
                return last;
            } else {
                // 将删除的元素放入堆顶
                data.set(0, last);
                // 自上向下调整堆
                shiftDown(0);
                // 返回堆顶元素
                return first;
            }
        }

        /**
         * 获取堆顶元素，但不删除
         *
         * @return 堆顶元素
         */
```

```
        public T getTop() {
            // 如果堆已经为空，就抛出异常
            if (data.isEmpty()) {
                throw new RuntimeException("The heap is empty.")
;
            }

            return data.get(0);
        }

        /**
         * 获取堆的大小
         *
         * @return 堆的大小
         */
        public int size() {
            return data.size();
        }

        /**
         * 判断堆是否为空
         *
         * @return 堆是否为空
         */
        public boolean isEmpty() {
            return data.isEmpty();
        }

        /**
         * 清空堆
         */
        public void clear() {
            data.clear();
        }

        /**
         * 获取堆中所有的数据
         *
         * @return 堆中所在的数据
         */
```

```
        public List<T> getData() {
            return data;
        }
    }

    /**
     * 升序比较器
     */
    private static class IncComparator implements Comparator<Int
eger> {

        @Override
        public int compare(Integer o1, Integer o2) {
            return o1 - o2;
        }
    }

    /**
     * 降序比较器
     */
    private static class DescComparator implements Comparator<In
teger> {

        @Override
        public int compare(Integer o1, Integer o2) {
            return o2 - o1;
        }
    }

    private static class DynamicArray {
        private Heap<Integer> max;
        private Heap<Integer> min;

        public DynamicArray() {
            max = new Heap<>(new IncComparator());
            min = new Heap<>(new DescComparator());
        }

        /**
         * 插入数据
```

```
     *
     * @param num 待插入的数据
     */
    public void insert(Integer num) {
        // 已经有偶数个数据了（可能没有数据）
        // 数据总数是偶数个时把新数据插入到小堆中
        if ((min.size() + max.size()) % 2 == 0) {
            // 大堆中有数据，并且插入的元素比大堆中的元素小
            if (max.size() > 0 && num < max.getTop()) {
                // 将num加入的大堆中去
                max.add(num);
                // 删除堆顶元素，大堆中的最大元素
                num = max.deleteTop();
            }

            // num插入到小堆中，当num小于大堆中的最大值进，
            // num就会变成大堆中的最大值，见上面的if操作
            // 如果num不小于大堆中的最大值，num就是自身
            min.add(num);
        }
        // 数据总数是奇数个时把新数据插入到大堆中
        else {
            // 小堆中有数据，并且插入的元素比小堆中的元素大
            if (min.size() > 0 && num > min.size()) {
                // 将num加入的小堆中去
                min.add(num);
                // 删除堆顶元素，小堆中的最小元素
                num = min.deleteTop();
            }
            // num插入到大堆中，当num大于小堆中的最小值进，
            // num就会变成小堆中的最小值，见上面的if操作
            // 如果num不大于大堆中的最小值，num就是自身
            max.add(num);
        }
    }

    public double getMedian() {
        int size = max.size() + min.size();

        if (size == 0) {
```

```
                throw new RuntimeException("No numbers are avail
able");
            }

            if ((size & 1) == 1) {
                return min.getTop();
            } else {
                return (max.getTop() + min.getTop()) / 2.0;
            }
        }
    }
}
```

一、题目

给定一个数组和滑动窗口的大小，请找出所有滑动窗口里的最大值。

举例说明

例如，如果输入数组{2,3,4,2,6,2,5,1}及滑动窗口的大小为3，那么一共存在6个滑动窗口，它们的最大值分别为{4,4,6,6,6,5}。

二、解题思路

如果采用蛮力法，这个问题似乎不难解决：可以扫描每一个滑动窗口的所有数字并找出其中的最大值。如果滑动窗口的大小为k，需要O(k)时间才能找出滑动窗口里的最大值。对于长度为n的输入数组，这个算法总的时间复杂度是O(nk)。

实际上一个滑动窗口可以看成是一个队列。当窗口滑动时，处于窗口的第一个数字被删除，同时在窗口的末尾添加一个新的数字。这符合队列的先进先出特性。如果能从队列中找出它的最大数，这个问题也就解决了。

但我们并不把滑动窗口的每个数值都存入队列中，而只把有可能成为滑动窗口最大值的数值存入到一个两端开口的队列。接着以输入数字{2,3,4,2,6,2,5,1}为例一步分析。

数组的第一个数字是2，把它存入队列中。第二个数字是3.由于它比前一个数字2大，因此2不可能成为滑动窗口中的最大值。2先从队列里删除，再把3存入到队列中。此时队列中只有一个数字3.针对第三个数字4的步骤类似，最终在队列中只剩下一个数字4.此时滑动窗口中已经有3个数字，而它的最大值4位于队列的头部。

接下来处理第四个数字2。2比队列中的数字4小。当4滑出窗口之后2还是有可能成为滑动窗口的最大值，因此把2存入队列的尾部。现在队列中有两个数字4和2，其中最大值4仍然位于队列的头部。

下一个数字是6.由于它比队列中已有的数字4和2都大，因此这时4和2已经不可能成为滑动窗口中的最大值。先把4和2从队列中删除，再把数字6存入队列。这个时候最大值6仍然位于队列的头部。

第六个数字是2.由于它比队列中已有的数字6小，所以2也存入队列的尾部。此时队列中有两个数字，其中最大值6位于队列的头部。

接下来的数字是5.在队列中已有的两个数字6和2里，2小于5，因此2不可能是一个滑动窗口的最大值，可以把它从队列的尾部删除。删除数字2之后，再把数字5存入队列。此时队列里剩下两个数字6和5，其中位于队列头部的是最大值6.

数组最后一个数字是1，把1存入队列的尾部。注意到位于队列头部的数字6是数组的第5个数字，此时的滑动窗口已经不包括这个数字了，因此应该把数字6从队列删除。

那么怎么知道滑动窗口是否包括一个数字？应该在队列里存入数字在数组里的下标，而不是数值。当一个数字的下标与当前处理的数字的下标之差大于或者等于滑动窗口的大小时，这个数字已经从滑动窗口中滑出，可以从队列中删除了。

三、解题代码

```
public class Test {
    private static List<Integer> maxInWindows(List<Integer> data
, int size) {
        List<Integer> windowMax = new LinkedList<>();

        // 条件检查
        if (data == null || size < 1 || data.size() < 1) {
            return windowMax;
        }

        Deque<Integer> idx = new LinkedList<>();

        // 窗口还没有被填满时，找最大值的索引
        for (int i = 0; i < size && i < data.size(); i++) {
            // 如果索引对应的值比之前存储的索引值对应的值大或者相等，就删
除之前存储的值
            while (!idx.isEmpty() && data.get(i) >= data.get(idx
.getLast())) {
                idx.removeLast();
            }

            //  添加索引
            idx.addLast(i);
        }
```

```
        // 窗口已经被填满了
        for (int i = size; i < data.size(); i++) {
            // 第一个窗口的最大值保存
            windowMax.add(data.get(idx.getFirst()));

            // 如果索引对应的值比之前存储的索引值对应的值大或者相等，就删
除之前存储的值
            while (!idx.isEmpty() && data.get(i) >= data.get(idx
.getLast())) {
                idx.removeLast();
            }

            // 删除已经滑出窗口的数据对应的下标
            if (!idx.isEmpty() && idx.getFirst() <= (i - size))
{
                idx.removeFirst();
            }

            // 可能的最大的下标索引入队
            idx.addLast(i);
        }

        // 最后一个窗口最大值入队
        windowMax.add(data.get(idx.getFirst()));

        return windowMax;

    }
}
```

一、题目

请设计一个函数，用来判断在一个矩阵中是否存在一条包含某字符串所有字符的路径。路径可以从矩阵中任意一格开始，每一步可以在矩阵中间向左、右、上、下移动一格。如果一条路径经过了矩阵的某一格，那么该路径不能再次进入该格子。

举例分析

例如在下面的3*4的矩阵中包含一条字符串”bcced”的路径。但矩阵中不包含字符串“abcb”的路径，因为字符串的第一个字符b占据了矩阵中的第一行第二格子之后，路径不能再次进入这个格子。

a b c e s f c s a d e e

二、解题思路

这是一个可以用回朔法解决的典型题。首先，在矩阵中任选一个格子作为路径的起点。除在矩阵边界上的格子之外，其他格子都有4个相邻的格子。重复这个过程直到路径上的所有字符都在矩阵中找到相应的位置。

由于回朔法的递归特性，路径可以被开成一个栈。当在矩阵中定位了路径中前n个字符的位置之后，在与第n个字符对应的格子的周围都没有找到第n+1个字符，这个时候只要在路径上回到第n-1个字符，重新定位第n个字符。

由于路径不能重复进入矩阵的格子，还需要定义和字符矩阵大小一样的布尔值矩阵，用来标识路径是否已经进入每个格子。

当矩阵中坐标为（row,col）的格子和路径字符串中下标为pathLength的字符一样时，从4个相邻的格子(row,col-1),(row-1,col),(row,col+1)以及(row+1,col)中去定位路径字符串中下标为pathLength+1的字符。

如果4个相邻的格子都没有匹配字符串中下标为pathLength+1的字符，表明当前路径字符串中下标为pathLength的字符在矩阵中的定位不正确，我们需要回到前一个字符(pathLength-1)，然后重新定位。

一直重复这个过程，直到路径字符串上所有字符都在矩阵中找到合适的位置

三、解题代码

```
public class Test {
    /**
     * 题目：请设计一个函数，用来判断在一个矩阵中是否存在一条包含某字符串所
有字符的路径。
     * 路径可以从矩阵中任意一格开始，每一步可以在矩阵中间向左、右、上、下移
动一格。
     * 如果一条路径经过了矩阵的某一格，那么该路径不能再次进入该格子。
     *
     * @param matrix 输入矩阵
     * @param rows   矩阵行数
     * @param cols   矩阵列数
     * @param str    要搜索的字符串
     * @return 是否找到 true是，false否
     */
    public static boolean hasPath(char[] matrix, int rows, int c
ols, char[] str) {
        // 参数校验
        if (matrix == null || matrix.length != rows * cols || st
r == null || str.length < 1) {
            return false;
        }

        // 变量初始化
        boolean[] visited = new boolean[rows * cols];
        for (int i = 0; i < visited.length; i++) {
            visited[i] = false;
        }

        // 记录结果的数组，
        int[] pathLength = {0};
        // 以每一个点为起始进行搜索
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                if (hasPathCore(matrix, rows, cols, str, visited
, i, j, pathLength)) {
                    return true;
                }
            }
        }
```

```
        return false;
    }

    /**
     * 回溯搜索算法
     *
     * @param matrix     输入矩阵
     * @param rows       矩阵行数
     * @param cols       矩阵列数
     * @param str        要搜索的字符串
     * @param visited    访问标记数组
     * @param row        当前处理的行号
     * @param col        当前处理的列号
     * @param pathLength 已经处理的str中字符个数
     * @return 是否找到 true是，false否
     */
    private static boolean hasPathCore(char[] matrix, int rows,
int cols, char[] str, boolean[] visited,
                                       int row, int col, int[] p
athLength) {

        if (pathLength[0] == str.length) {
            return true;
        }

        boolean hasPath = false;

        // 判断位置是否合法
        if (row >= 0 && row < rows
                && col >= 0 && col < cols
                && matrix[row * cols + col] == str[pathLength[0]
]
                && !visited[row * cols + col]) {

            visited[row * cols + col] = true;
            pathLength[0]++;

            // 按左上右下进行回溯
            hasPath = hasPathCore(matrix, rows, cols, str, visit
```

```
ed, row, col - 1, pathLength)
                    || hasPathCore(matrix, rows, cols, str, visi
ted, row - 1, col, pathLength)
                    || hasPathCore(matrix, rows, cols, str, visi
ted, row, col + 1, pathLength)
                    || hasPathCore(matrix, rows, cols, str, visi
ted, row + 1, col, pathLength);

            if (!hasPath) {
                pathLength[0]--;
                visited[row * cols + col] = false;
            }

        }

        return hasPath;
    }
}
```

一、题目

地上有个m行n列的方格。一个机器人从坐标(0,0)的格子开始移动，它每一次可以向左、右、上、下移动一格，但不能进入行坐标和列坐标的数位之和大于k的格子。

举例分析

例如，当k为18时，机器人能够进入方格(35,37)，因为3+5+3+7=18.但它不能进入方格(35,38)，因为3+5+3+8=19.请问该机器人能够达到多少格子？

二、解题思路

这个方格也可以看出一个m*n的矩阵。同样在这个矩阵中，除边界上的格子之外其他格子都有四个相邻的格子。

机器人从坐标(0,0)开始移动。当它准备进入坐标为(i,j)的格子时，通过检查坐标的数位和来判断机器人是否能够进入。如果机器人能够进入坐标为(i,j)的格子，我们接着再判断它能否进入四个相邻的格子(i,j-1)、(i-1,j),(i,j+1)和(i+1,j)。

三、解题代码

```
public class Test {
    /**
     * 题目：地上有个m行n列的方格。一个机器人从坐标(0,0)的格子开始移动，
     * 它每一次可以向左、右、上、下移动一格，但不能进入行坐标和列坐标的数
     * 位之和大于k的格子。例如，当k为18时，机器人能够进入方格(35,37)，
     * 因为3+5+3+7=18.但它不能进入方格(35,38)，因为3+5+3+8=19.
     * 请问该机器人能够达到多少格子？
     *
     * @param threshold 约束值
     * @param rows      方格的行数
     * @param cols      方格的列数
     * @return 最多可走的方格
     */
    public static int movingCount(int threshold, int rows, int c
```

```
ols) {
        // 参数校验
        if (threshold < 0 || rows < 1 || cols < 1) {
            return 0;
        }

        // 变量初始化
        boolean[] visited = new boolean[rows * cols];
        for (int i = 0; i < visited.length; i++) {
            visited[i] = false;
        }

        return movingCountCore(threshold, rows, cols, 0, 0, visi
ted);
    }

    /**
     * 递归回溯方法
     *
     * @param threshold 约束值
     * @param rows      方格的行数
     * @param cols      方格的列数
     * @param row       当前处理的行号
     * @param col       当前处理的列号
     * @param visited   访问标记数组
     * @return 最多可走的方格
     */
    private static int movingCountCore(int threshold, int rows,
int cols,
                                       int row, int col, boolean
[] visited) {

        int count = 0;

        if (check(threshold, rows, cols, row, col, visited)) {
            visited[row * cols + col] = true;
            count = 1
                    + movingCountCore(threshold, rows, cols, row
 - 1, col, visited)
                    + movingCountCore(threshold, rows, cols, row
```

```
, col - 1, visited)
                    + movingCountCore(threshold, rows, cols, row
 + 1, col, visited)
                    + movingCountCore(threshold, rows, cols, row
, col + 1, visited);
        }

        return count;
    }

    /**
     * 断机器人能否进入坐标为(row, col)的方格
     *
     * @param threshold 约束值
     * @param rows      方格的行数
     * @param cols      方格的列数
     * @param row       当前处理的行号
     * @param col       当前处理的列号
     * @param visited   访问标记数组
     * @return 是否可以进入，true是，false否
     */
    private static boolean check(int threshold, int rows, int co
ls,
                                 int row, int col, boolean[] vis
ited) {
        return col >= 0 && col < cols
                && row >= 0 && row < rows
                && !visited[row * cols + col]
                && (getDigitSum(col) + getDigitSum(row) <= thres
hold);
    }

    /**
     * 一个数字的数位之和
     *
     * @param number 数字
     * @return 数字的数位之和
     */
    private static int getDigitSum(int number) {
        int result = 0;
```

```
        while (number > 0) {
            result += (number % 10);
            number /= 10;
        }

        return result;
    }
}
```

一、前言

LeetCode这部分内容的算法比较难，自己能力有限，刷的题目也比较少，这部分内容就面试及笔试出现的频率整理出一部分内容，熟悉相关解题方法，争取在笔试中能够AC。

整理内容主要包括以下几个部分：

- 数组
- 字符串
- 链表
- 动态规划
- 贪心算法

二、目录

一、动态规划

1. 简介

动态规划的本质，是对问题状态的定义和状态转移方程的定义。

> **dynamic programming** is a method for solving a complex problem by **breaking it down into a collection of simpler subproblems**.

动态规划是通过拆分问题，定义问题状态和状态之间的关系，使得问题能够以递推（或者说分治）的方式去解决。

以下介绍，大多都是在说递推的求解方法，但如何拆分问题，才是动态规划的核心。

而拆分问题，靠的就是状态的定义和状态转移方程的定义。

2. 状态的定义

首先想说大家千万不要被下面的数学式吓到，这里只涉及到了函数相关的知识。

> 给定一个数列，长度为N， 求这个数列的最长上升（递增）子数列（LIS）的长度. 以 1 7 2 8 3 4 为例。 这个数列的最长递增子数列是 1 2 3 4，长度为4； 次长的长度为3， 包括 1 7 8; 1 2 3 等.

要解决这个问题，我们首先要定义这个问题和这个问题的子问题。 有人可能会问了，题目都已经在这了，我们还需定义这个问题吗？需要，原因就是这个问题在字面上看，找不出子问题，而没有子问题，这个题目就没办法解决。

> 给定一个数列，长度为N，设
>
> $$F_k$$
>
> 为：以数列中第k项结尾的最长递增子序列的长度. 求 中的最大值.

显然，这个新问题与原问题等价。 而对于

$$F_k$$

来讲，□都是

$$F_k$$

的子问题：因为以第k项结尾的最长递增子序列（下称LIS），包含着以第

$$1..k-1$$

中某项结尾的LIS。

上述的新问题

$$F_k$$

也可以叫做状态，定义中的“

$$F_k$$

为数列中第k项结尾的LIS的长度"，就叫做对状态的定义。之所以把

$$F_k$$

做“状态”而不是“问题” ，一是因为避免跟原问题中“问题”混淆，二是因为这个新问题是数学化定义的。

对状态的定义只有一种吗？当然不是

> 给定一个数列，长度为N， 设$F{i,k}$为： 在前i项中的，长度为k的最长递增子序列中，最后一位的最小值. 1<=k<=N. 若在前i项中，不存在长度为k的最长递增子序列，则$F{i,k}$为正无穷. 求最大的x，使得$F_{N,k}$不为正无穷。

这个新定义与原问题的等价性也不难证明，请读者体会一下。 上述的$F{i,k}$就是状态，定义中的“$F{i,k}$为：在前i项中，长度为k的最长递增子序列中，最后一位的最小值”就是对状态的定义。

3. 状态转移方程

上述状态定义好之后，状态和状态之间的关系式，就叫做状态转移方程。

> 设
>
> $$F_k$$
>
> 为：以数列中第k项结尾的最长递增子序列的长度.

设A为题中数列，状态转移方程为：

用文字解释一下是：

以第k项结尾的LIS的长度是：保证第i项比第k项小的情况下，以第i项结尾的LIS长度加一的最大值，取遍i的所有值（i小于k）。

第二种定义：

> 设$F_{i,k}$为：在数列前i项中，长度为k的递增子序列中，最后一位的最小值

设A为题中数列，状态转移方程为：

（边界情况需要分类讨论较多，在此不列出，需要根据状态定义导出边界情况。）

大家套着定义读一下公式就可以了，应该不难理解，就是有点绕。

这里可以看出，这里的状态转移方程，就是定义了问题和子问题之间的关系。

可以看出，状态转移方程就是带有条件的递推式。

二、目录

本部分内容整理一些LeetCode中关于动态规划的常见问题及Java解决方案，供大家学习动态规划。

- Distinct Subsequences
- Longest Common Subsequence
- Longest Increasing Subsequence
- Best Time to Buy and Sell Stock
- Maximum Subarray
- Maximum Product Subarray
- Longest Palindromic Substring
- BackPack
- Maximal Square
- Stone Game

一、题目

> Given a string **S** and a string **T**, count the number of distinct subsequences of **T** in **S**.A subsequence of a string is a new string which is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (ie, `"ACE"` is a subsequence of `"ABCDE"` while `"AEC"` is not).
>
> Here is an example:
>
> **S** = `"rabbbit"`, **T** = `"rabbit"`
>
> Return `3`.

给定两个字符串S和T，求S有多少个不同的子串与T相同。S的子串定义为在S中任意去掉0个或者多个字符形成的串。

二、解题思路

动态规划，设`dp[i][j]` 是从字符串S[0...i]中删除几个字符得到字符串T[0...j]的不同的删除方法种类，动态规划方程如下

- 如果S[i] = T[j], `dp[i][j] = dp[i-1][j-1]+dp[i-1`][j]
- 如果S[i] 不等于 T[j], `dp[i][j] = dp[i-1][j]`
- 初始条件：当T为空字符串时，从任意的S删除几个字符得到T的方法为1
- `dp[0][0] = 1`; // T和S都是空串.

  `dp[1 ... S.length() - 1][0] = 1;` // T是空串，S只有一种子序列匹配。

  `dp[0][1 ... T.length() - 1] = 0;` // S是空串，T不是空串，S没有子序列匹配。

三、解题代码

```
public int numDistincts(String S, String T)
{
  int[][] table = new int[S.length() + 1][T.length() + 1];

  for (int i = 0; i < S.length(); i++)
      table[i][0] = 1;

  for (int i = 1; i <= S.length(); i++) {
      for (int j = 1; j <= T.length(); j++) {
          if (S.charAt(i - 1) == T.charAt(j - 1)) {
              table[i][j] += table[i - 1][j] + table[i - 1][j - 1
];
          } else {
              table[i][j] += table[i - 1][j];
          }
      }
  }

  return table[S.length()][T.length()];
}
```

一、题目

> Given two strings, find the longest common subsequence (LCS).
>
> Your code should return the length of LCS.
>
> Have you met this question in a real interview?
>
> Yes
>
> Example
>
> For "ABCD" and "EDCA", the LCS is "A" (or "D", "C"), return 1.
>
> For "ABCD" and "EACB", the LCS is "AC", return 2.

求最长公共子序列的数目，注意这里的子序列可以不是连续序列。

二、解题思路

求『最长』类的题目往往与动态规划有点关系，这里是两个字符串，故应为双序列动态规划。

这道题的状态很容易找，不妨先试试以 `f[i][j]` 表示字符串 A 的前 `i` 位和字符串 B 的前 `j` 位的最长公共子序列数目，那么接下来试试寻找其状态转移方程。从实际例子 `ABCD` 和 `EDCA` 出发，首先初始化 `f` 的长度为字符串长度加1，那么有 `f[0][0] = 0`, `f[0][*] = 0`, `f[*][0] = 0`, 最后应该返回 `f[lenA][lenB]`. 即 f 中索引与字符串索引对应(字符串索引从1开始算起)，那么在A 的第一个字符与 B 的第一个字符相等时， `f[1][1] = 1 + f[0][0]`, 否则 `f[1][1] = max(f[0][1], f[1][0])` 。

推而广之，也就意味着若 `A[i] == B[j]`, 则分别去掉这两个字符后，原 LCS 数目减一，那为什么一定是1而不是0或者2呢？因为不管公共子序列是以哪个字符结尾，在 `A[i] == B[j]` 时 LCS 最多只能增加1. 而在 `A[i] != B[j]` 时，由于 `A[i]` 或者 `B[j]` 不可能同时出现在最终的 LCS 中，故这个问题可进一步缩小， `f[i][j] = max(f[i - 1][j], f[i][j - 1])`. 需要注意的是这种状态转移方程只依赖最终的 LCS 数目，而不依赖于公共子序列到底是以第几个索引结束。

三、解题代码

```
public class Solution {
    /**
     * @param A, B: Two strings.
     * @return: The length of longest common subsequence of A an
d B.
     */
    public int longestCommonSubsequence(String A, String B) {
        if (A == null || A.length() == 0) return 0;
        if (B == null || B.length() == 0) return 0;

        int lenA = A.length();
        int lenB = B.length();
        int[][] lcs = new int[1 + lenA][1 + lenB];

        for (int i = 1; i < 1 + lenA; i++) {
            for (int j = 1; j < 1 + lenB; j++) {
                if (A.charAt(i - 1) == B.charAt(j - 1)) {
                    lcs[i][j] = 1 + lcs[i - 1][j - 1];
                } else {
                    lcs[i][j] = Math.max(lcs[i - 1][j], lcs[i][j
 - 1]);
                }
            }
        }

        return lcs[lenA][lenB];
    }
}
```

一、题目

> Given a sequence of integers, find the longest increasing subsequence (LIS).
>
> You code should return the length of the LIS.
>
> Example For [5, 4, 1, 2, 3], the LIS is [1, 2, 3], return 3
>
> For [4, 2, 4, 5, 3, 7], the LIS is [4, 4, 5, 7], return 4

二、解题思路

方案一：动态规划 时间复杂度**O(n*n)**

`dp[i]` 表示以i结尾的子序列中LIS的长度。然后我用 `dp[j](0<=j<i)` 来表示在i之前的LIS的长度。然后我们可以看到，只有当 `a[i]>a[j]` 的时候，我们需要进行判断，是否将a[i]加入到dp[j]当中。为了保证我们每次加入都是得到一个最优的LIS，有两点需要注意：第一，每一次，a[i]都应当加入最大的那个dp[j]，保证局部性质最优，也就是我们需要找到 `max(dp[j](0<=j<i))` ；第二，每一次加入之后，我们都应当更新dp[j]的值，显然， `dp[i]=dp[j]+1` 。 如果写成递推公式，我们可以得到 `dp[i]=max(dp[j](0<=j<i))+(a[i]>a[j]?1:0)` 。

方案二：二分搜索 时间复杂度**O(nlogn)**

开一个栈，每次取栈顶元素top和读到的元素temp做比较，如果temp > top 则将temp入栈；如果temp < top则二分查找栈中的比temp大的第1个数，并用temp替换它。 最长序列长度即为栈的大小top。

这也是很好理解的，对于x和y，如果x < y且Stack[y] < Stack[x],用Stack[x]替换Stack[y]，此时的最长序列长度没有改变但序列Q的"潜力"增大了。

举例：原序列为1，5，8，3，6，7

栈为1，5，8，此时读到3，用3替换5，得到1，3，8； 再读6，用6替换8，得到1，3，6；再读7，得到最终栈为1，3，6，7。最长递增子序列为长度4。

三、解题代码

方案一：

```
public int longestIncreasingSubsequence(int[] nums) {
        int []f = new int[nums.length];
        int max = 0;
        for (int i = 0; i < nums.length; i++) {
            f[i] = 1;
            for (int j = 0; j < i; j++) {
                if (nums[j] < nums[i]) {
                    f[i] = f[i] > f[j] + 1 ? f[i] : f[j] + 1;
                }
            }
            if (f[i] > max) {
                max = f[i];
            }
        }
        return max;
    }
```

方案二：

```
public int findLongest(int[] A, int n) {
        int length = A.length;
        int[] B = new int[length];
        B[0] = A[0];
        int end = 0;
        for (int i = 1; i < length; ++i) {
            // 如果当前数比B中最后一个数还大，直接添加
            if (A[i] >= B[end]) { B[++end] = A[i]; continue;
            }
            // 否则，需要先找到替换位置
            int pos = findInsertPos(B, A[i], 0, end);
            B[pos] = A[i];
        }
        for (int i = 0; i < B.length; ++i) {
            System.out.println(B[i]);
        }
        return end+ 1; }
    /**
     * 二分查找第一个大于等于n的位置
     */
    private int findInsertPos(int[] B, int n, int start, int end)
 {
        while (start < end) {
            int mid = start + (end - start) / 2;// 直接使用(high
+ low) / 2 可能导致溢出
            if (B[mid] < n) {
                start = mid + 1;
            } else if (B[mid] > n) {
                end = mid ;
            } else {
                return mid;
            }
        }
        return start;
    }
```

1.1 题目

> Say you have an array for which the ith element is the price of a given stock on day i.
>
> If you were only permitted to complete at most one transaction (ie, buy one and sell one share of the stock), design an algorithm to find the maximum profit.

这是卖股票的一个题目，一个数组prices，其中prices[i]表示第i天股票的价格。根据题意我们知道只能进行一次交易，但需要获得最大的利润。

1.2 解题思路

我们需要在最低价买入，最高价卖出，当然买入一定要在卖出之前。

对于这一题，还是比较简单的，我们只需要遍历一次数组，通过一个变量记录当前最低价格，同时算出此次交易利润，并与当前最大值比较就可以了。

1.3 解题代码

```
public class Solution {
    public int maxProfit(int[] prices) {
        if (prices == null || prices.length == 0) {
            return 0;
        }

        int min = Integer.MAX_VALUE;  //just remember the smalle
st price
        int profit = 0;
        for (int i : prices) {
            min = i < min ? i : min;
            profit = (i - min) > profit ? i - min : profit;
        }

        return profit;
    }
}
```

2.1 题目

> Say you have an array for which the ith element is the price of a given stock on day i.
>
> Design an algorithm to find the maximum profit. You may complete as many transactions as you like (ie, buy one and sell one share of the stock multiple times). However, you may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

假设有一个数组，它的第i个元素是一个给定的股票在第i天的价格。设计一个算法来找到最大的利润。你可以完成尽可能多的交易(多次买卖股票)。然而,你不能同时参与多个交易(你必须在再次购买前出售股票)。

2.2 解题思路

因为不限制交易次数，我们在第i天买入，如果发现i + 1天比i高，那么就可以累加到利润里面。

2.3 解题代码

```
public class Solution {
    public int maxProfit(int[] prices) {
        int profit = 0;
        for (int i = 0; i < prices.length - 1; i++) {
            int diff = prices[i+1] - prices[i];
            if (diff > 0) {
                profit += diff;
            }
        }
        return profit;
    }
}
```

3.1 题目

> Say you have an array for which the ith element is the price of a given stock on day i.
>
> Design an algorithm to find the maximum profit. You may complete at most two transactions.
>
> Note: You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

假设你有一个数组，它的第i个元素是一支给定的股票在第i天的价格。设计一个算法来找到最大的利润。你最多可以完成两笔交易。然而,你不能同时参与多个交易(你必须在再次购买前出售股票)。

3.2 解题思路

最多允许两次不相交的交易，也就意味着这两次交易间存在某一分界线，考虑到可只交易一次，也可交易零次，故分界线的变化范围为第一天至最后一天，只需考虑分界线两边各自的最大利润，最后选出利润和最大的即可。

这种方法抽象之后则为首先将 [1,n] 拆分为 [1,i] 和 [i+1,n], 参考卖股票系列的第一题计算各自区间内的最大利润即可。[1,i] 区间的最大利润很好算，但是如何计算 [i+1,n] 区间的最大利润值呢？难道需要重复 n 次才能得到？注意到区间的右侧 n 是个不变值，我们从 [1, i] 计算最大利润是更新波谷的值，那么我们可否逆序计算最大利润呢？这时候就需要更新记录波峰的值了

3.3 解题代码

```
public class Solution {
    /**
     * @param prices: Given an integer array
     * @return: Maximum profit
     */
    public int maxProfit(int[] prices) {
        if (prices == null || prices.length <= 1) return 0;

        // get profit in the front of prices
        int[] profitFront = new int[prices.length];
        profitFront[0] = 0;
        for (int i = 1, valley = prices[0]; i < prices.length; i
++) {
            profitFront[i] = Math.max(profitFront[i - 1], prices
[i] - valley);
            valley = Math.min(valley, prices[i]);
        }
        // get profit in the back of prices, (i, n)
        int[] profitBack = new int[prices.length];
        profitBack[prices.length - 1] = 0;
        for (int i = prices.length - 2, peak = prices[prices.len
gth - 1]; i >= 0; i--) {
            profitBack[i] = Math.max(profitBack[i + 1], peak - p
rices[i]);
            peak = Math.max(peak, prices[i]);
        }
        // add the profit front and back
        int profit = 0;
        for (int i = 0; i < prices.length; i++) {
            profit = Math.max(profit, profitFront[i] + profitBac
k[i]);
        }

        return profit;
    }
};
```

4.1 题目

> Say you have an array for which the ith element is the price of a given stock on day i.
>
> Design an algorithm to find the maximum profit. You may complete at most k transactions.
>
> Example
>
> Given prices = [4,4,6,1,1,4,2,5], and k = 2, return 6.
>
> Note
>
> You may not engage in multiple transactions at the same time (i.e., you must sell the stock before you buy again).
>
> Challenge
>
> O(nk) time.

题目和上面一样，就是变成要求交易k次，时间复杂度O(nk)。

4.2 解题思路

我们仍然使用动态规划来完成。我们维护两种量，一个是当前到达第i天可以最多进行j次交易，最好的利润是多少（`global[i][j]`），另一个是当前到达第i天，最多可进行j次交易，并且最后一次交易在当天卖出的最好的利润是多少（`local[i][j]`）。下面我们来看递推式，全局的比较简单，

`global[i][j]=max(local[i][j],global[i-1][j])`，

也就是去当前局部最好的，和过往全局最好的中大的那个（因为最后一次交易如果包含当前天一定在局部最好的里面，否则一定在过往全局最优的里面）。

全局（到达第i天进行j次交易的最大收益） = max{局部（在第i天交易后，恰好满足j次交易），全局（到达第i-1天时已经满足j次交易）}

对于局部变量的维护，递推式是

`local[i][j]=max(global[i-1][j-1]+max(diff,0),local[i-1][j]+diff)`，

也就是看两个量，第一个是全局到i-1天进行j-1次交易，然后加上今天的交易，如果今天是赚钱的话（也就是前面只要j-1次交易，最后一次交易取当前天），第二个量则是取local第i-1天j次交易，然后加上今天的差值（这里因为 `local[i-1][j]` 比如包含第i-1天卖出的交易，所以现在变成第i天卖出，并不会增加交易次数，而且这里无论diff是不是大于0都一定要加上，因为否则就不满足local[i][j]必须在最后一天卖出的条件了）。

局部（在第i天交易后，总共交易了j次） = max{情况2，情况1}

情况1：在第i-1天时，恰好已经交易了j次（ `local[i-1][j]` ），那么如果i-1天到i天再交易一次：即在第i-1天买入，第i天卖出（diff），则这不并不会增加交易次数！【例如我在第一天买入，第二天卖出；然后第二天又买入，第三天再卖出的行为 和 第一天买入，第三天卖出 的效果是一样的，其实只进行了一次交易！因为有连续性】 情况2：第i-1天后，共交易了j-1次（ `global[i-1][j-1]` ），因此为了满足“第i天过后共进行了j次交易，且第i天必须进行交易”的条件：我们可以选择1：在第i-1天买入，然后再第i天卖出（diff），或者选择在第i天买入，然后同样在第i天卖出（收益为0）。

上面的算法中对于天数需要一次扫描，而每次要对交易次数进行递推式求解，所以时间复杂度是O(n*k)，如果是最多进行两次交易，那么复杂度还是O(n)。空间上只需要维护当天数据皆可以，所以是O(k)，当k=2，则是O(1)。

补充：这道题还有一个陷阱，就是当k大于天数时，其实就退化成 Best Time to Buy and Sell Stock II 了。

4.3 解题代码

```java
public class Solution {
    /**
     * @param k: An integer
     * @param prices: Given an integer array
     * @return: Maximum profit
     */
    public int maxProfit(int k, int[] prices) {
        if (prices == null || prices.length < 2) {
            return 0;
        }
        int days = prices.length;
```

```
        if (days <= k) {
            return maxProfit2(prices);
        }
        // local[i][j] 表示前i天，至多进行j次交易，第i天必须sell的最大
获益
        int[][] local = new int[days][k + 1];
        // global[i][j] 表示前i天，至多进行j次交易，第i天可以不sell的最
大获益
        int[][] global = new int[days][k + 1];

        for (int i = 1; i < days; i++) {
            int diff = prices[i] - prices[i - 1];
            for (int j = 1; j <= k; j++) {
                local[i][j] = Math.max(global[i - 1][j-1] + Math
.max(diff, 0),
                                       local[i - 1][j] + diff);
                global[i][j] = Math.max(global[i - 1][j], local[
i][j]);
            }
        }
        return global[days - 1][k];
    }

    public int maxProfit2(int[] prices) {
        int maxProfit = 0;
        for (int i = 1; i < prices.length; i++) {
            if (prices[i] > prices[i-1]) {
                maxProfit += prices[i] - prices[i-1];
            }
        }
        return maxProfit;
    }
};
```

一、题目

> Find the contiguous subarray within an array (containing at least one number) which has the largest sum.
>
> For example, given the array [−2,1,−3,4,−1,2,1,−5,4], the contiguous subarray [4,−1,2,1] has the largest sum = 6.
>
> More practice: If you have figured out the O(n) solution, try coding another solution using the divide and conquer approach, which is more subtle.

二、解题思路

方案一

典型的DP题：

1. 状态dp[i]：以A[i]为最后一个数的所有max subarray的和。
2. 通项公式：dp[i] = dp[i-1]<=0 ? dp[i] : dp[i-1]+A[i]
3. 由于dp[i]仅取决于dp[i-1]，所以可以仅用一个变量来保存前一个状态，而节省内存。

方案二

虽然这道题目用dp解起来很简单，但是题目说了，问我们能不能采用divide and conquer的方法解答，也就是二分法。

假设数组A[left, right]存在最大区间，mid = (left + right) / 2，那么无非就是三中情况：

1. 最大值在A[left, mid - 1]里面
2. 最大值在A[mid + 1, right]里面
3. 最大值跨过了mid，也就是我们需要计算[left, mid - 1]区间的最大值，以及[mid + 1, right]的最大值，然后加上mid，三者之和就是总的最大值

我们可以看到，对于1和2，我们通过递归可以很方便的求解，然后在同第3的结果比较，就是得到的最大值。

三、解题代码

方案一

```
public class Solution {
    /**
     * @param nums: A list of integers
     * @return: A integer indicate the sum of max subarray
     */
    public int maxSubArray(int[] A) {
        int n = A.length;
        int[] dp = new int[n]; //dp[i] means the maximum subarra
y ending with A[i];
        dp[0] = A[0];
        int max = dp[0];

        for(int i = 1; i < n; i++){
            dp[i] = A[i] + (dp[i - 1] > 0 ? dp[i - 1] : 0);
            max = Math.max(max, dp[i]);
        }

        return max;
    }
}
```

方案二

```
public class Solution {
    public int maxSubArray(int[] A) {
        int maxSum = Integer.MIN_VALUE;
        return findMaxSub(A, 0, A.length - 1, maxSum);
    }

    // recursive to find max sum
    // may appear on the left or right part, or across mid(from
left to right)
    public int findMaxSub(int[] A, int left, int right, int maxS
um) {
        if(left > right)    return Integer.MIN_VALUE;
```

```
        // get max sub sum from both left and right cases
        int mid = (left + right) / 2;
        int leftMax = findMaxSub(A, left, mid - 1, maxSum);
        int rightMax = findMaxSub(A, mid + 1, right, maxSum);
        maxSum = Math.max(maxSum, Math.max(leftMax, rightMax));

        // get max sum of this range (case: across mid)
        // so need to expend to both left and right using mid as
 center
        // mid -> left
        int sum = 0, midLeftMax = 0;
        for(int i = mid - 1; i >= left; i--) {
            sum += A[i];
            if(sum > midLeftMax)    midLeftMax = sum;
        }
        // mid -> right
        int midRightMax = 0; sum = 0;
        for(int i = mid + 1; i <= right; i++) {
            sum += A[i];
            if(sum > midRightMax)    midRightMax = sum;
        }

        // get the max value from the left, right and across mid
        maxSum = Math.max(maxSum, midLeftMax + midRightMax + A[m
id]);

        return maxSum;
    }
}
```

一、题目

> Find the contiguous subarray within an array (containing at least one number) which has the largest product.
>
> For example, given the array [2,3,-2,4], the contiguous subarray [2,3] has the largest product = 6.

在一个数组中，寻找一个连续子数组使得成绩最大。

二、解题思路

这题是求数组中子区间的最大乘积，对于乘法，我们需要注意，负数乘以负数，会变成正数，所以解这题的时候我们需要维护两个变量，当前的最大值，以及最小值，最小值可能为负数，但没准下一步乘以一个负数，当前的最大值就变成最小值，而最小值则变成最大值了。

DP的四要素

- 状态：
  - `max_product[i]` : 以nums[i]结尾的max subarray product
  - `min_product[i]` : 以nums[i]结尾的min subarray product
- 方程：
  - `max_product[i] = getMax(max_product[i-1] * nums[i], min_product[i-1] * nums[i], nums[i])`
  - `min_product[i] = getMin(max_product[i-1] * nums[i], min_product[i-1] * nums[i], nums[i])`
- 初始化：
  - `max_product[0] = min_product[0] = nums[0]`
- 结果：
  - 每次循环中 max_product[i] 的最大值

三、解题代码

```
public class Solution {
    /**
     * @param nums: an array of integers
     * @return: an integer
     */
    public int maxProduct(List<Integer> nums) {
        int[] max = new int[nums.size()];
        int[] min = new int[nums.size()];

        min[0] = max[0] = nums.get(0);
        int result = nums.get(0);
        for (int i = 1; i < nums.size(); i++) {
            min[i] = max[i] = nums.get(i);
            if (nums.get(i) > 0) {
                max[i] = Math.max(max[i], max[i - 1] * nums.get(
i));
                min[i] = Math.min(min[i], min[i - 1] * nums.get(
i));
            } else if (nums.get(i) < 0) {
                max[i] = Math.max(max[i], min[i - 1] * nums.get(
i));
                min[i] = Math.min(min[i], max[i - 1] * nums.get(
i));
            }

            result = Math.max(result, max[i]);
        }

        return result;
    }
}
```

一、题目

> Given a string S, find the longest palindromic substring in S.
>
> You may assume that the maximum length of S is 1000, and there exists one unique longest palindromic substring.
>
> **Example**
>
> Given the string = `"abcdzdcab"`, return `"cdzdc"`.
>
> **Challenge**
>
> O(n2) time is acceptable. Can you do it in O(n) time.

求一个字符串中的最长回文子串。

二、解题思路

区间类动态规划

Time O(n^2), Space O(n^2)

用 `dp[i][j]` 来存DP的状态，需要较多的额外空间: Space O(n^2)

DP的4个要素

- 状态：
  - `dp[i][j]` : s.charAt(i)到s.charAt(j)是否构成一个Palindrome
- 转移方程：
  - `dp[i][j] = s.charAt(i) == s.charAt(j) && (j - i <= 2 || dp[i + 1][j - 1])`
- 初始化：
  - `dp[i][j] = true` when `j - i <= 2`
- 结果：
  - 找 `maxLen = j - i + 1;` ，并得到相应longest substring： `longest = s.substring(i, j + 1);`

中心扩展

这种方法基本思想是遍历数组，以其中的1个元素或者2个元素作为palindrome的中心，通过辅助函数，寻找能拓展得到的最长子字符串。外层循环 O(n)，内层循环O(n)，因此时间复杂度 Time O(n^2)，相比动态规划二维数组存状态的方法，因为只需要存最长palindrome子字符串本身，这里空间更优化：Space O(1)。

三、解题代码

区间DP，Time O(n^2) Space O(n^2)

```
public class Solution {
    /**
     * @param s input string
     * @return the longest palindromic substring
     */
    public String longestPalindrome(String s) {
        if(s == null || s.length() <= 1) {
            return s;
        }

        int len = s.length();
        int maxLen = 1;
        boolean [][] dp = new boolean[len][len];

        String longest = null;
        for(int k = 0; k < s.length(); k++){
            for(int i = 0; i < len - k; i++){
                int j = i + k;
                if(s.charAt(i) == s.charAt(j) && (j - i <= 2 ||
 dp[i + 1][j - 1])){
                    dp[i][j] = true;

                    if(j - i + 1 > maxLen){
                       maxLen = j - i + 1;
                       longest = s.substring(i, j + 1);
                    }
                }
            }
        }

        return longest;
    }
}
```

Time O(n^2) Space O(1)

```
public class Solution {
    /**
     * @param s input string
```

```
     * @return the longest palindromic substring
     */
    public String longestPalindrome(String s) {
            if (s.isEmpty()) {
                return null;
            }

            if (s.length() == 1) {
                return s;
            }

            String longest = s.substring(0, 1);
            for (int i = 0; i < s.length(); i++) {
                // get longest palindrome with center of i
                String tmp = helper(s, i, i);
                if (tmp.length() > longest.length()) {
                    longest = tmp;
                }

                // get longest palindrome with center of i, i+1
                tmp = helper(s, i, i + 1);
                if (tmp.length() > longest.length()) {
                    longest = tmp;
                }
            }

            return longest;
    }

    // Given a center, either one letter or two letter,
    // Find longest palindrome
    public String helper(String s, int begin, int end) {
        while (begin >= 0 && end <= s.length() - 1 && s.charAt(b
egin) == s.charAt(end)) {
            begin--;
            end++;
        }
        return s.substring(begin + 1, end);
    }
}
```

1.1 题目

> Given n items with size A[i], an integer m denotes the size of a backpack. How full you can fill this backpack?
>
> NoteYou
>
> can not divide any item into small pieces
>
> .Example
>
> If we have 4 items with size [2, 3, 5, 7], the backpack size is 11, we can select 2, 3 and 5, so that the max size we can fill this backpack is 10.
>
> If the backpack size is 12. we can select [2, 3, 7] so that we can fulfill the backpack.
>
> You function should return the max size we can fill in the given backpack.

在n个物品中挑选若干物品装入背包，最多能装多满？假设背包的大小为m，每个物品的大小为A[i]。

1.2 解题思路

本题是典型的01背包问题，每种类型的物品最多只能选择一件。

1. State: `dp[i][S]` 表示前i个物品，取出一些能否组成和为S体积的背包
2. Function: `f[i][S] = f[i-1][S - A[i]] or f[i-1][S]` (A[i]表示第i个物品的大小)

转移方程想得到f[i][S]前i个物品取出一些物品想组成S体积的背包。 那么可以从两个状态转换得到。

（1） `f[i-1][S - A[i]]` 放入第i个物品，并且前i-1个物品能否取出一些组成和为 `S-A[i]` 体积大小的背包。

（2） `f[i-1][S]` 不放入第i个物品， 并且前i-1个物品能否取出一些组成和为S 体积大小的背包。

1. Intialize: `f[1…n][0]` = true; `f[0][1... m]` = false

初始化 `f[1...n][0]` 表示前1...n个物品，取出一些能否组成和为0 大小的背包始终为真。

其他初始化为假

1. Answer: 寻找使 `f[n][S]` 值为true的最大的S. （S的取值范围1到m）

1.3 解题代码

```
public class Solution {
    /**
     * @param m: An integer m denotes the size of a backpack
     * @param A: Given n items with size A[i]
     * @return: The maximum size
     */
    public int backPack(int m, int[] A) {
        boolean f[][] = new boolean[A.length + 1][m + 1];
        for (int i = 0; i <= A.length; i++) {
            for (int j = 0; j <= m; j++) {
                f[i][j] = false;
            }
        }
        f[0][0] = true;
        for (int i = 1; i <= A.length; i++) {
            for (int j = 0; j <= m; j++) {
                f[i][j] = f[i - 1][j];
                if (j >= A[i-1] && f[i-1][j - A[i-1]]) {
                    f[i][j] = true;
                }
            } // for j
        } // for i

        for (int i = m; i >= 0; i--) {
          for (int j = A.length;j>=0;j--){
              if (f[j][i]) {
                return i;
            }
          }
        }

        return 0;
    }
}
```

2.1 题目

> Given n items with size A[i] and value V[i], and a backpack with size m. What's the maximum value can you put into the backpack?
>
> Note
>
> You cannot divide item into small pieces and the total size of items you choose should smaller or equal to m.
>
> Example
>
> Given 4 items with size [2, 3, 5, 7] and value [1, 5, 2, 4], and a backpack with size 10.
>
> The maximum value is 9.

两个数组，一个表示体积，另一个表示价值，给定一个容积为m的背包，求背包装入物品的最大价值。

2.2 解题思路

首先定义状态 K(i,w) 为前 i 个物品放入size为 w 的背包中所获得的最大价值，则相应的状态转移方程为： $K(i,w)=max\{K(i-1,w),K(i-1,w-wi)+vi\}$

2.3 解题代码

```
public class Solution {
    /**
     * @param m: An integer m denotes the size of a backpack
     * @param A & V: Given n items with size A[i] and value V[i]
     * @return: The maximum value
     */

    public int backPackII(int m, int[] A, int V[]) {
        // write your code here
        int[][] dp = new int[A.length + 1][m + 1];
        for(int i = 0; i <= A.length; i++){
            for(int j = 0; j <= m; j++){
                if(i == 0 || j == 0){
                    dp[i][j] = 0;
                }
                else if(A[i-1] > j){
                    dp[i][j] = dp[(i-1)][j];
                }
                else{
                    dp[i][j] = Math.max(dp[(i-1)][j], dp[(i-1)][
j-A[i-1]] + V[i-1]);
                }
            }
        }
        return dp[A.length][m];
    }
}
```

3.1 题目

> Given n kind of items with size Ai and value Vi(each item has an infinite number available) and a backpack with size m. What's the maximum value can you put into the backpack?
>
> Notice
>
> You cannot divide item into small pieces and the total size of items you choose should smaller or equal to m.
>
> Example
>
> Given 4 items with size [2, 3, 5, 7] and value [1, 5, 2, 4], and a backpack with size 10. The maximum value is 15.

这道题相比上题变为：重复选择+最大价值。

3.2 解题思路

和01背包问题很类似

状态转移方程

不放A[i]

`f[i][j] =f[i-1][j]`

放A[j]

可放多个设为k，

k = j/A[i]

`f[i][j] = f[i-1][j- ki*A[i]] + ki*A[i]` 0<=ki<=k 取最大值 0<=ki*A[i] <=m

3.3 解题代码

```
public class Solution {
    /**
     * 多重背包问题
     * 总体积是m，每个小物品的体积是A[i]
     *
     * @param m: An integer m denotes the size of a backpack
     * @param A: Given n items with size A[i] 0 开始的 A是
     * @return: The maximum size
     */
    public int backPackIII(int m, int[] A) {
        // write your code here
        int[][] P = new int[A.length+1][m+1];// P[i][j] 前i个物品
放在j的空间中的最大价值
        for(int i = 0;i< A.length; i++){
            for(int j = m;j>=1;j--){
                if(j>=A[i]){
                    int k = j/A[i];// 该物品最大可以放k个
                    while(k>=0){
                        if(j>=A[i]*k){
                            P[i+1][j] =Math.max(P[i+1][j], P[i][
j-k*A[i]] + k*A[i]);
                        }
                        k--;
                    }
                }

                else
                    P[i+1][j] = Math.max(P[i][j],P[i+1][j]);
            }
        }
        return P[A.length][m];
    }
}
```

一、题目

> Given a 2D binary matrix filled with 0's and 1's, find the largest square containing all 1's and return its area.
>
> **Example**
>
> For example, given the following matrix:
>
> ```
> 1 0 1 0 0
> 1 0 1 1 1
> 1 1 1 1 1
> 1 0 0 1 0
> ```
>
> Return 4.

给定一个二维矩阵，其中元素值为0或1,找出最大的一个正方形，使得其元素都为1,返回其面积。

二、解题思路

当我们判断以某个点为正方形右下角时最大的正方形时，那它的上方，左方和左上方三个点也一定是某个正方形的右下角，否则该点为右下角的正方形最大就是它自己了。

这是定性的判断，那具体的最大正方形边长呢？我们知道，该点为右下角的正方形的最大边长，最多比它的上方，左方和左上方为右下角的正方形的边长多1，最好的情况是是它的上方，左方和左上方为右下角的正方形的大小都一样的，这样加上该点就可以构成一个更大的正方形。

但如果它的上方，左方和左上方为右下角的正方形的大小不一样，合起来就会缺了某个角落，这时候只能取那三个正方形中最小的正方形的边长加1了。假设 `dp[i][j]` 表示以i,j为右下角的正方形的最大边长，则有

```
dp[i][j] = min(dp[i-1][j-1], dp[i-1][j], dp[i][j-1]) + 1
```

当然，如果这个点在原矩阵中本身就是0的话，那 `dp[i][j]` 肯定就是0了。

三、解题代码

```
public class Solution {
    public int maximalSquare(char[][] matrix) {
        if(matrix.length == 0) return 0;
        int m = matrix.length, n = matrix[0].length;
        int max = 0;
        int[][] dp = new int[m][n];
        // 第一列赋值
        for(int i = 0; i < m; i++){
            dp[i][0] = matrix[i][0] - '0';
            max = Math.max(max, dp[i][0]);
        }
        // 第一行赋值
        for(int i = 0; i < n; i++){
            dp[0][i] = matrix[0][i] - '0';
            max = Math.max(max, dp[0][i]);
        }
        // 递推
        for(int i = 1; i < m; i++){
            for(int j = 1; j < n; j++){
                dp[i][j] = matrix[i][j] == '1' ? Math.min(dp[i-1
][j-1], Math.min(dp[i-1][j], dp[i][j-1])) + 1 : 0;
                max = Math.max(max, dp[i][j]);
            }
        }
        return max * max;
    }
}
```

一、题目

> There is a stone game.At the beginning of the game the player picks `n` piles of stones in a line.
>
> The goal is to merge the stones in one pile observing the following rules:
>
> 1. At each step of the game, the player can merge two adjacent piles to a new pile.
> 2. The score is the number of stones in the new pile.
>
> You are to determine the **minimum** of the total score.
>
> **Example**
>
> For `[4, 1, 1, 4]`, in the best solution, the total score is `18`:
>
> ```
> 1. Merge second and third piles => [4, 2, 4], score +2
> 2. Merge the first two piles => [6, 4]，score +6
> 3. Merge the last two piles => [10], score +10
> ```
>
> Other two examples:
>
> `[1, 1, 1, 1]` return `8` `[4, 4, 5, 9]` return `43`

一堆石头，每个石头代表一个值。每次可以合并两个相邻的石头，得分是合并后的和。一直合并，同时累计得分，直到变成一个石头，并求出得分最小的值。

二、解题思路

这道题可用DP解。

`dp[i][j]` 表示合并i到j的石头需要的最小代价。

转移函数：

`dp[i][j]=dp[i][k]+dp[k+1][j]+sum[i][j]` （i<=k<j）。即合并i－j的代价为合并左边部分的代价＋合并右边部分的代价＋合并左右部分的代价（即i－j所有元素的总和）。找到使 `dp[i][j]` 最小的k。

DP四要素

- State:
  - `dp[i][j]` 表示把第i到第j个石子合并到一起的最小花费
- Function:
  - 预处理 `sum[i][j]` 表示i到j所有石子价值和
  - `dp[i][j] = min(dp[i][k]+dp[k+1][j]+sum[i][j])` 对于所有 `k` 属于 `{i,j}`
- Intialize:
  - for each i
    - `dp[i][i] = 0`
- Answer:
  - `dp[0][n-1]`

区间型DP，利用二维数组下标表示下标范围。需要注意的是对状态转移方程的理解，也就是对每一种分割方式进行遍历。

三、解题代码

```
public class Solution {
    /**
     * @param A an integer array
     * @return an integer
     */
    public int stoneGame(int[] A) {
        // Write your code here
        // DP
        if(A == null || A.length == 0){
            return 0;
        }

        int n = A.length;
        int[][] sum = new int[n][n];
        for(int i = 0; i < n; i++){
            sum[i][i] = A[i];
            for(int j = i + 1; j < n; j++){
                sum[i][j] = sum[i][j - 1] + A[j];
            }
```

```
        }

        int[][] dp = new int[n][n];
        for(int i = 0; i < n; i++){
            dp[i][i] = 0;
        }

        for(int len = 2; len <= n; len++){
            for(int i = 0; i + len - 1 < n; i++){
                int j = i + len - 1;
                int min = Integer.MAX_VALUE;
                for(int k = i; k < j; k++){
                    min = Math.min(min, dp[i][k] + dp[k + 1][j])
;
                }
                dp[i][j] = min + sum[i][j];
            }
        }

        return dp[0][n - 1];
    }
}
```

一、数组

数组是较为简单的数据结构，它占据一块连续的内存，并按照顺序存储数据。数组需要事先知道容量大小，然后根据大小分配存储空间，所以数组的空间利用率不高。数组有很好的查找效率，能在O(1)内找到元素。所以我们可以基于数组实现简单的hash表，提高查找效率。

关于数组是面试中常考的一种数据结构，此类的相关题目相对简单，一般通过数字规律，指针，动态规划等方法来解决。

二、目录

一、题目

> Given an array nums of integers and an int k, partition the array (i.e move the elements in "nums") such that:
>
> - All elements < k are moved to the left
> - All elements >= k are moved to the right
> - Return the partitioning index, i.e the first index i nums[i] >= k.
>
> *Notice*
>
> You should do really partition in array nums instead of just counting the numbers of integers smaller than k.
>
> If all elements in nums are smaller than k, then return nums.length

二、解题思路

根据给定的k，也就是类似于Quick Sort中的pivot，将array从两头进行缩进，时间复杂度 O(n)

三、解题代码

```
public class Solution {
    private void swap(int i, int j, int[] arr) {
        int tmp = arr[i];
        arr[i] = arr[j];
        arr[j] = tmp;
    }
    /**
     *@param nums: The integer array you should partition
     *@param k: As description
     *return: The index after partition
     */
    public int partitionArray(int[] nums, int k) {

        int pl = 0;
        int pr = nums.length - 1;
        while (pl <= pr) {
            while (pl <= pr && nums[pl] < k) {
                pl++;
            }
            while (pl <= pr && nums[pr] >= k) {
                pr--;
            }
            if (pl <= pr) {
                swap(pl, pr, nums);
                pl++;
                pr--;
            }
        }
        return pl;
    }
}
```

一、题目

> Given an integer array, find a subarray where the sum of numbers is zero. Your code should return the index of the first number and the index of the last number.
>
> **Example**
>
> Given `[-3, 1, 2, -3, 4]`, return `[0, 2]` or `[1, 3]`.

给定一个整数数组，找到和为零的子数组。你的代码应该返回满足要求的子数组的起始位置和结束位置

二、解题思路

记录每一个位置的sum，存入HashMap中，如果某一个sum已经出现过，那么说明中间的subarray的sum为0. 时间复杂度O(n)，空间复杂度O(n)

三、解题代码

```
public class Solution {
    /**
     * @param nums: A list of integers
     * @return: A list of integers includes the index of the fir
st number
     *          and the index of the last number
     */
    public ArrayList<Integer> subarraySum(int[] nums) {
        // write your code here

        int len = nums.length;

        ArrayList<Integer> ans = new ArrayList<Integer>();
        HashMap<Integer, Integer> map = new HashMap<Integer, Int
eger>();

        map.put(0, -1);

        int sum = 0;
        for (int i = 0; i < len; i++) {
            sum += nums[i];

            if (map.containsKey(sum)) {
                ans.add(map.get(sum) + 1);
                ans.add(i);
                return ans;
            }

            map.put(sum, i);
        }

        return ans;
    }
}
```

一、题目

> Given a non-negative number represented as an array of digits, plus one to the number.
>
> The digits are stored such that the most significant digit is at the head of the list.
>
> Example
>
> Given [1,2,3] which represents 123, return [1,2,4].
>
> Given [9,9,9] which represents 999, return [1,0,0,0].

给一个包含非负整数的数组，其中每个值代表该位数的值，对这个数加1。

二、解题思路

1. 数组的最后一个数是个位数，所以从后面开始读，个位数+1后，如果有进位，存储进位值，没有直接存储。
2. 处理十位数，如果个位数有进位，十位数+1,在判断十位数有没有进位。
3. 重复上面的动作直到没有进位。

三、解题代码

```
public class Solution {

    public int[] plusOne(int[] digits) {
        int carries = 1;
        for(int i = digits.length-1; i>=0 && carries > 0; i--){
// fast break when carries equals zero
            int sum = digits[i] + carries;
            digits[i] = sum % 10;
            carries = sum / 10;
        }
        if(carries == 0)
            return digits;

        int[] rst = new int[digits.length+1];
        rst[0] = 1;
        for(int i=1; i< rst.length; i++){
            rst[i] = digits[i-1];
        }
        return rst;
    }
}
```

一、题目

> Determine whether an integer is a palindrome. Do this without extra space.

给定一个数字，要求判断这个数字是否为回文数字. 比如121就是回文数字，122就不是回文数字.

二、解题思路

题目要求只能用O(1)的空间，所以不能考虑把它转化为字符串然后reverse比较的方法。

基本思路是每次去第一位和最后一位，如果不相同则返回false，否则继续直到位数为0。

需要注意的点:

1. 负数不是回文数字.
2. 0是回文数字.

三、解题代码

```
public boolean isPalindrome(int x) {
    if(x<0)
        return false;
    int div = 1;
    while(div<=x/10)
        div *= 10;
    while(x>0)
    {
        if(x/div!=x%10)
            return false;
        x = (x%div)/10;
        div /= 100;
    }
    return true;
}
```

一、题目

> Given an array of integers, return indices of the two numbers such that they add up to a specific target.
>
> You may assume that each input would have exactly one solution.
>
> Example:
>
> Given nums = [2, 7, 11, 15], target = 9, Because nums[0] + nums[1] = 2 + 7 = 9, return [0, 1].

给定一个整型数组，找出能相加起来等于一个特定目标数字的两个数。

二、解题思路

用hashmap，hashmap是内部存储方式为哈希表的map结构。遍历数组，其中key存放目标值减去当前值，value存放对应索引。如果在遍历过程中发现map中存在与当前值相等的key，则返回结果。

三、解题代码

```
public class Solution {
    /*
     * @param numbers : An array of Integer
     * @param target : target = numbers[index1] + numbers[index2
]
     * @return : [index1 + 1, index2 + 1] (index1 < index2)
         numbers=[2, 7, 11, 15],  target=9
         return [1, 2]
     */
    public int[] twoSum(int[] numbers, int target) {
        HashMap<Integer,Integer> map = new HashMap<>();

        for (int i = 0; i < numbers.length; i++) {
            if (map.get(numbers[i]) != null) {
                int[] result = {map.get(numbers[i]) + 1, i + 1};
                return result;
            }
            map.put(target - numbers[i], i);
        }

        int[] result = {};
        return result;
    }
}
```

一、字符串

关于字符串的题目，在面试和笔试中都有很多出现。主要包括以下几个方面：

1. 大数问题。
2. 字符串的翻转（全部翻转、部分翻转），拼接等问题。
3. 字符串的模式匹配，找重复子串、公共前缀、回文问题。

以上几个方面的题目在剑指offer以及leetcode动态规划方面都有出现过。本部分主要整理了几个比较典型常考的题目。

二、目录

一、题目

> Given a string containing only digits, restore it by returning all possible valid IP address combinations.
>
> For example: Given `"25525511135"`,
>
> return `["255.255.11.135", "255.255.111.35"]`. (Order does not matter)

给一个由数字组成的字符串。求出其可能恢复为的所有IP地址。

注意 ：中间IP位置不能以0开始，0.01.01.1非法，应该是0.0.101.1或者0.0.10.11

二、解题思路

方法一：

直接三种循环暴力求解

方法二：

深度搜索，回溯

三、解题代码

方法一

```
public class Solution {
    /**
     * @param s the IP string
     * @return All possible valid IP addresses
     */
    public ArrayList<String> restoreIpAddresses(String s) {
        ArrayList<String> res = new ArrayList<String>();
        int len = s.length();
        for(int i = 1; i<4 && i<len-2; i++){
            for(int j = i+1; j<i+4 && j<len-1; j++){
                for(int k = j+1; k<j+4 && k<len; k++){
                    String s1 = s.substring(0,i), s2 = s.substri
ng(i,j), s3 = s.substring(j,k), s4 = s.substring(k,len);
                    if(isValid(s1) && isValid(s2) && isValid(s3)
 && isValid(s4)){
                        res.add(s1+"."+s2+"."+s3+"."+s4);
                    }
                }
            }
        }
        return res;
    }
    public boolean isValid(String s){
        if(s.length()>3 || s.length()==0 || (s.charAt(0)=='0' &&
 s.length()>1) || Integer.parseInt(s)>255)
            return false;
        return true;
    }
}
```

方法二

```
public class Solution {
    /**
     * @param s the IP string
     * @return All possible valid IP addresses
     */
    public ArrayList<String> restoreIpAddresses(String s) {
        ArrayList<String> result = new ArrayList<String>();
```

```
        ArrayList<String> list = new ArrayList<String>();

        if(s.length() <4 || s.length() > 12)
            return result;

        helper(result, list, s , 0);
        return result;
    }

    public void helper(ArrayList<String> result, ArrayList<Strin
g> list, String s, int start){
        if(list.size() == 4){
            if(start != s.length())
                return;

            StringBuffer sb = new StringBuffer();
            for(String tmp: list){
                sb.append(tmp);
                sb.append(".");
            }
            sb.deleteCharAt(sb.length()-1);
            result.add(sb.toString());
            return;
        }

        for(int i=start; i<s.length() && i < start+3; i++){
            String tmp = s.substring(start, i+1);
            if(isvalid(tmp)){
                list.add(tmp);
                helper(result, list, s, i+1);
                list.remove(list.size()-1);
            }
        }
    }

    private boolean isvalid(String s){
        if(s.charAt(0) == '0')
            return s.equals("0"); // to eliminate cases like "00
", "10"
        int digit = Integer.valueOf(s);
```

```
        return digit >= 0 && digit <= 255;
    }
}
```

一、题目

> Given a string and an offset, rotate string by offset. (rotate from left to right)
>
> Example
>
> Given `"abcdefg"`.
>
> offset=0 => "abcdefg" offset=1 => "gabcdef" offset=2 => "fgabcde" offset=3 => "efgabcd"
>
> Challenge
>
> Rotate in-place with O(1) extra memory.

给定一个字符串和一个偏移量，根据偏移量旋转字符串(从左向右旋转)

二、解题思路

常见的翻转法应用题，仔细观察规律可知翻转的分割点在从数组末尾数起的offset位置。先翻转前半部分，随后翻转后半部分，最后整体翻转。

三、解题代码

```
public class Solution {
    /*
     * param A: A string
     * param offset: Rotate string with offset.
     * return: Rotated string.
     */
    public char[] rotateString(char[] A, int offset) {
        if (A == null || A.length == 0) {
            return A;
        }

        int len = A.length;
        offset %= len;
        reverse(A, 0, len - offset - 1);
        reverse(A, len - offset, len - 1);
        reverse(A, 0, len - 1);

        return A;
    }

    private void reverse(char[] str, int start, int end) {
        while (start < end) {
            char temp = str[start];
            str[start] = str[end];
            str[end] = temp;
            start++;
            end--;
        }
    }
}
```

一、题目

> Given a string, determine if it is a palindrome, considering only alphanumeric characters and ignoring cases.
>
> For example, `"A man, a plan, a canal: Panama"` is a palindrome. `"race a car"` is *not* a palindrome.
>
> **Note:**
>
> Have you consider that the string might be empty? This is a good question to ask during an interview.
>
> For the purpose of this problem, we define empty string as valid palindrome.

判断一个字符串是不是回文串。

二、解题思路

字符串的回文判断问题，由于字符串可随机访问，故逐个比较首尾字符是否相等最为便利，即常见的『两根指针』技法。

两步走：

1. 找到最左边和最右边的第一个合法字符(字母或者字符)
2. 一致转换为小写进行比较

三、解题代码

```
public class Solution {
    public boolean isPalindrome(String s) {
        if (s == null || s.trim().isEmpty()) {
            return true;
        }

        int l = 0, r = s.length() - 1;
        while (l < r) {
            if(!Character.isLetterOrDigit(s.charAt(l))) {
                l++;
                continue;
            }
            if(!Character.isLetterOrDigit(s.charAt(r))) {
                r--;
                continue;
            }
            if (Character.toLowerCase(s.charAt(l)) == Character.
toLowerCase(s.charAt(r))) {
                l++;
                r--;
            } else {
                return false;
            }
        }

        return true;
    }
}
```

一、题目

> Given a string *s* consists of upper/lower-case alphabets and empty space characters `' '`, return the length of last word in the string.
>
> If the last word does not exist, return 0.
>
> **Note:** A word is defined as a character sequence consists of non-space characters only.
>
> For example, Given *s* = `"Hello World"`, return `5`.

给定一个字符串， 包含大小写字母、空格`' '`，请返回其最后一个单词的长度。

如果不存在最后一个单词，请返回 `0` 。

二、解题思路

关键点在于确定最后一个字符串之前的空格，此外还需要考虑末尾空格这一特殊情况。从最后往前扫描。

三、解题代码

```
public class Solution {
    public int lengthOfLastWord(String s) {
        if (s == null || s.isEmpty()) return 0;

        int len = 0;
        for (int i = s.length() - 1; i >= 0; i--) {
            if (s.charAt(i) == ' ') {
                if (len > 0) return len;
            } else {
                len++;
            }
        }

        return len;
    }
}
```

一、链表

链表是面试中十分容易考到的题目，一般代码比较短，而且考查面试者的思维全面性和写无bug代码的能力。在写链表的题目时，建议画出示意图，并把头结点、尾节点这些特殊的结点考虑在内。

常考题型如下:

题一、 给定单链表，检测是否有环

题二、 给定两个单链表(head1, head2)，检测两个链表是否有交点，如果有返回第一个交点。

题三、 给定单链表(head)，如果有环的话请返回从头结点进入环的第一个节点。

题四、只给定单链表中某个结点p(并非最后一个结点，即p->next!=NULL)指针，删除该结点。

题五、只给定单链表中某个结点p(非空结点)，在p前面插入一个结点。

题六、给定单链表头结点，删除链表中倒数第k个结点

题七、复杂链表复制

题八、两个不交叉的有序链表的合并

题九、链表翻转（包括全翻转，部分翻转，分段翻转）（递归或非递归实现）

题十、实现链表排序的一种算法

题十一、删除有序单链表中重复的元素

题十二、用链表模拟大整数加法运算

链表可以说是面试高频必问知识点，而关于链表的题目也比较固定。以上题目在剑指offer上大多出现过。本部分主要整理了几个比较典型常考的题目。

二、目录

- Merge Two Sorted Lists
- LRU Cache
- Remove Linked List Elements

一、题目

> Given a sorted linked list, delete all duplicates such that each element appear only *once*.
>
> For example, Given `1->1->2`, return `1->2`. Given `1->1->2->3->3`, return `1->2->3`.

给定一个排序链表，删除所有重复的元素每个元素只留下一个。

二、解题思路

遍历之，遇到当前节点和下一节点的值相同时，删除下一节点，并将当前节点 `next` 值指向下一个节点的 `next`，当前节点首先保持不变，直到相邻节点的值不等时才移动到下一节点。

三、解题代码

```
/**
 * Definition for ListNode
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param ListNode head is the head of the linked list
     * @return: ListNode head of linked list
     */
    public static ListNode deleteDuplicates(ListNode head) {
        ListNode curr = head;
        while (curr != null) {
            while (curr.next != null && curr.val == curr.next.va
l) {
                curr.next = curr.next.next;
            }
            curr = curr.next;
        }

        return head;
    }
}
```

一、题目

> Given a linked list and a value *x*, partition it such that all nodes less than *x* come before nodes greater than or equal to *x*.
>
> You should preserve the original relative order of the nodes in each of the two partitions.
>
> For example, Given `1->4->3->2->5->2` and *x* = 3, return `1->2->2->4->3->5`.

给定一个单链表和数值x，划分链表使得所有小于x的节点排在大于等于x的节点之前。

你应该保留两部分内链表节点原有的相对顺序。

二、解题思路

依据题意，是要根据值x对链表进行分割操作，具体是指将所有小于x的节点放到不小于x的节点之前，咋一看和快速排序的分割有些类似，但是这个题的不同之处在于只要求将小于x的节点放到前面，而并不要求对元素进行排序。

这种分割的题使用两路指针即可轻松解决。左边指针指向小于x的节点，右边指针指向不小于x的节点。由于左右头节点不确定，我们可以使用两个dummy节点。

三、解题代码

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ListNode partition(ListNode head, int x) {
        ListNode leftDummy = new ListNode(0);
        ListNode leftCurr = leftDummy;
        ListNode rightDummy = new ListNode(0);
        ListNode rightCurr = rightDummy;

        ListNode runner = head;
        while (runner != null) {
            if (runner.val < x) {
                leftCurr.next = runner;
                leftCurr = leftCurr.next;
            } else {
                rightCurr.next = runner;
                rightCurr = rightCurr.next;
            }
            runner = runner.next;
        }

        // cut off ListNode after rightCurr to avoid cylic
        rightCurr.next = null;
        leftCurr.next = rightDummy.next;

        return leftDummy.next;
    }
}
```

一、题目

> Merge two sorted (ascending) linked lists and return it as a new sorted list. The new sorted list should be made by splicing together the nodes of the two lists and sorted in ascending order.
>
> Example
>
> Given `1->3->8->11->15->null` , `2->null` , return `1->2->3->8->11->15->null` .

将两个排序链表合并为一个新的排序链表

二、解题思路

只需要从头开始比较已排序的两个链表，新链表指针每次指向值小的节点，依次比较下去，最后，当其中一个链表到达了末尾，我们只需要把新链表指针指向另一个没有到末尾的链表此时的指针即可。

三、解题代码

```
/**
 * Definition for ListNode.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int val) {
 *         this.val = val;
 *         this.next = null;
 *     }
 * }
 */
public class Solution {
    /**
     * @param ListNode l1 is the head of the linked list
     * @param ListNode l2 is the head of the linked list
     * @return: ListNode head of linked list
     */
    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        ListNode dummy = new ListNode(0);
        ListNode curr = dummy;

        while ((l1 != null) && (l2 != null)) {
            if (l1.val > l2.val) {
                curr.next = l2;
                l2 = l2.next;
            } else {
                curr.next = l1;
                l1 = l1.next;
            }
            curr = curr.next;
        }

        // link to non-null list
        curr.next = (l1 != null) ? l1 : l2;

        return dummy.next;
    }
}
```

一、题目

> Design and implement a data structure for Least Recently Used (LRU) cache. It should support the following operations: `get` and `set`.
>
> `get(key)` - Get the value (will always be positive) of the key if the key exists in the cache, otherwise return -1.
>
> `set(key, value)` - Set or insert the value if the key is not already present. When the cache reached its capacity, it should invalidate the least recently used item before inserting a new item.

为最近最少使用（LRU）缓存策略设计一个数据结构，它应该支持以下操作：获取数据（**get**）和写入数据（**set**）。

获取数据**get(key)**：如果缓存中存在key，则获取其数据值（通常是正数），否则返回-1。

写入数据**set(key, value)**：如果key还没有在缓存中，则写入其数据值。当缓存达到上限，它应该在写入新数据之前删除最近最少使用的数据用来腾出空闲位置。

二、解题思路

双向链表加哈希表

缓存讲究的就是快，所以我们必须做到O(1)的获取速度，这样看来只有哈希表可以胜任。但是哈希表无序的，我们没办法在缓存满时，将最早更新的元素给删去。那么是否有一种数据结构，可以将先进的元素先出呢？那就是队列。所以我们将元素存在队列中，并用一个哈希表记录下键值和元素的映射，就可以做到O(1)获取速度，和先进先出的效果。然而，当我们获取一个元素时，还需要把这个元素再次放到队列头，这个元素可能在队列的任意位置，可是队列并不支持对任意位置的增删操作。而最适合对任意位置增删操作的数据结构又是什么呢？是链表。我可以用链表来实现一个队列，这样就同时拥有链表和队列的特性了。不过，如果仅用单链表的话，在任意位置删除一个节点还是很麻烦的，要么记录下该节点的上一个节点，要么遍历一遍。所以双向链表是最好的选择。我们用双向链表实现一个队列用来记录每个元素的顺序，用一个哈希表来记录键和值的关系，就行了。

三、解题代码

```
public class Solution {
    private int capacity;
    private HashMap<Integer, Node> map = new HashMap<>();
    private Node head = new Node(-1, -1), tail = new Node(-1, -1
);

    private class Node {
        Node prev, next;
        int val, key;

        public Node(int key, int val) {
            this.val = val;
            this.key = key;
            prev = null;
            next = null;
        }

//          @Override
//          public String toString() {
//              return "(" + key + ", " + val + ") " + "last:"
//                      + (prev == null ? "null" : "node");
//          }
    }

    public Solution(int capacity) {
        this.capacity = capacity;
        tail.prev = head;
        head.next = tail;
    }

    public int get(int key) {
        if (!map.containsKey(key)) {
            return -1;
        }
        // remove current
        Node currentNode = map.get(key);
        currentNode.prev.next = currentNode.next;
```

```
            currentNode.next.prev = currentNode.prev;

            // move current to tail;
            moveToTail(currentNode);

            return map.get(key).val;
        }

        public void set(int key, int value) {
            if (get(key) != -1) {
                map.get(key).val = value;
                return;
            }
            if (map.size() == capacity) {
                map.remove(head.next.key);
                head.next = head.next.next;
                head.next.prev = head;
            }
            Node insert = new Node(key, value);
            map.put(key, insert);
            moveToTail(insert);
        }

        private void moveToTail(Node current) {
            current.prev = tail.prev;
            tail.prev = current;
            current.prev.next = current;
            current.next = tail;
        }
    }
```

一、题目

> Remove all elements from a linked list of integers that have value `val`.
>
> Example
>
> Given `1->2->3->3->4->5->3`, val = 3, you should return the list as `1->2->4->5`

删除链表中等于给定值 `val` 的所有节点。

二、解题思路

删除链表中指定值，找到其前一个节点即可，将 next 指向下一个节点即可

三、解题代码

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    /**
     * @param head a ListNode
     * @param val an integer
     * @return a ListNode
     */
    public ListNode removeElements(ListNode head, int val) {
        ListNode dummy = new ListNode(0);
        dummy.next = head;
        ListNode curr = dummy;
        while (curr.next != null) {
            if (curr.next.val == val) {
                curr.next = curr.next.next;
            } else {
                curr = curr.next;
            }
        }

        return dummy.next;
    }
}
```

一、贪心算法

贪心法，又称贪心算法、贪婪算法、或称贪婪法，是一种在每一步选择中都采取在当前状态下最好或最优（即最有利）的选择，从而希望导致结果是最好或最优的算法。

贪心算法在有最优子结构的问题中尤为有效。最优子结构的意思是局部最优解能决定全局最优解。简单地说，问题能够分解成子问题来解决，子问题的最优解能递推到最终问题的最优解。

贪心算法与动态规划的不同在于它对每个子问题的解决方案都做出选择，不能回退。动态规划则会保存以前的运算结果，并根据以前的结果对当前进行选择，有回退功能。

贪心法可以解决一些最优化问题，如：求图中的最小生成树、求哈夫曼编码......对于其他问题，贪心法一般不能得到我们所要求的答案。一旦一个问题可以通过贪心法来解决，那么贪心法一般是解决这个问题的最好办法。由于贪心法的高效性以及其所求得的答案比较接近最优结果，贪心法也可以用作辅助算法或者直接解决一些要求结果不特别精确的问题。

二、目录

1.1 题目

> Given an array of non-negative integers, you are initially positioned at the first index of the array.
>
> Each element in the array represents your maximum jump length at that position.
>
> Determine if you are able to reach the last index.
>
> For example: A = [2,3,1,1,4], return true.
>
> A = [3,2,1,0,4], return false.

给出一个非负整数数组，你最初定位在数组的第一个位置。

数组中的每个元素代表你在那个位置可以跳跃的最大长度。

判断你是否能到达数组的最后一个位置。

1.2 解题思路

注意题目中A[i]表示的是在位置i，“最大”的跳跃距离，而并不是指在位置i只能跳A[i]的距离。所以当跳到位置i后，能达到的最大的距离至少是i+A[i]。用greedy来解，记录一个当前能达到的最远距离maxIndex：

1. 能跳到位置i的条件：i<=maxIndex。
2. 一旦跳到i，则maxIndex = max(maxIndex, i+A[i])。
3. 能跳到最后一个位置n-1的条件是：maxIndex >= n-1

1.3 解题代码

```
public class Solution {
    public boolean canJump(int[] A) {
        // think it as merging n intervals
        if (A == null || A.length == 0) {
            return false;
        }
        int farthest = A[0];
        for (int i = 1; i < A.length; i++) {
            if (i <= farthest && A[i] + i >= farthest) {
                farthest = A[i] + i;
            }
        }
        return farthest >= A.length - 1;
    }
}
```

2.1 题目

> Given an array of non-negative integers, you are initially positioned at the first index of the array.
>
> Each element in the array represents your maximum jump length at that position.
>
> Your goal is to reach the last index in the minimum number of jumps.
>
> For example: Given array A = [2,3,1,1,4]
>
> The minimum number of jumps to reach the last index is 2. (Jump 1 step from index 0 to 1, then 3 steps to the last index.)

给出一个非负整数数组，你最初定位在数组的第一个位置。

数组中的每个元素代表你在那个位置可以跳跃的最大长度。

你的目标是使用最少的跳跃次数到达数组的最后一个位置。

2.2 解题思路

同样可以用greedy解决。与I不同的是，求的不是对每个i，从A[0:i]能跳到的最远距离；而是计算跳了k次后能达到的最远距离，这里的通项公式为：

d[k] = max(i+A[i]) d[k-2] < i <= d[k-1]

2.3 解题代码

```
public class Solution {
    public int jump(int[] A) {
        if (A == null || A.length == 0) {
            return -1;
        }
        int start = 0, end = 0, jumps = 0;
        while (end < A.length - 1) {
            jumps++;
            int farthest = end;
            for (int i = start; i <= end; i++) {
                if (A[i] + i > farthest) {
                    farthest = A[i] + i;
                }
            }
            start = end + 1;
            end = farthest;
        }
        return jumps;
    }
}
```

一、题目

> There are N gas stations along a circular route, where the amount of gas at station i is gas[i].
>
> You have a car with an unlimited gas tank and it costs cost[i] of gas to travel from station i to its next station (i+1). You begin the journey with an empty tank at one of the gas stations.
>
> Return the starting gas station's index if you can travel around the circuit once, otherwise return -1.
>
> Note: The solution is guaranteed to be unique.

在一条环路上有 *N* 个加油站，其中第 *i* 个加油站有汽油 `gas[i]` ，并且从第*i*个加油站前往第*i*+1个加油站需要消耗汽油 `cost[i]` 。

你有一辆油箱容量无限大的汽车，现在要从某一个加油站出发绕环路一周，一开始油箱为空。

求可环绕环路一周时出发的加油站的编号，若不存在环绕一周的方案，则返回 `-1` 。

二、解题思路

首先我们可以得到所有油站的油量totalGas，以及总里程需要消耗的油量totalCost，如果totalCost大于totalGas，那么铁定不能够走完整个里程。

如果totalGas大于totalCost了，那么就能走完整个里程了，假设现在我们到达了第i个油站，这时候还剩余的油量为sum，如果 sum + gas[i] - cost[i]小于0，我们无法走到下一个油站，所以起点一定不在第i个以及之前的油站里面（都铁定走不到第i + 1号油站），起点只能在i + 1后者后面。

三、解题代码

```
public class Solution {
    public int canCompleteCircuit(int[] gas, int[] cost) {
        if (gas == null || cost == null || gas.length == 0 || co
st.length == 0) {
            return -1;
        }

        int sum = 0;
        int total = 0;
        int index = -1;

        for(int i = 0; i<gas.length; i++) {
            sum += gas[i] - cost[i];
            total += gas[i] - cost[i];
            if(sum < 0) {
                index = i;
                sum = 0;
            }
        }
        return total < 0 ? -1 : index + 1;
        // index should be updated here for cases ([5], [4]);
        // total < 0 is for case [2], [2]
    }
}
```

一、题目

> There are N children standing in a line. Each child is assigned a rating value.
>
> You are giving candies to these children subjected to the following requirements:
>
> Each child must have at least one candy. Children with a higher rating get more candies than their neighbors. What is the minimum candies you must give?

有 *N* 个小孩站成一列。每个小孩有一个评级。

按照以下要求，给小孩分糖果：

- 每个小孩至少得到一颗糖果。
- 评级越高的小孩可以比他相邻的两个小孩得到更多的糖果。

需最少准备多少糖果？

二、解题思路

首先我们会给每个小朋友一颗糖果，然后从左到右，假设第i个小孩的等级比第i - 1个小孩高，那么第i的小孩的糖果数量就是第i - 1个小孩糖果数量在加一。再我们从右到左，如果第i个小孩的等级大于第i + 1个小孩的，同时第i个小孩此时的糖果数量小于第i + 1的小孩，那么第i个小孩的糖果数量就是第i + 1个小孩的糖果数量加一。

三、解题代码

```
public class Solution {
    public int candy(int[] ratings) {
        if(ratings == null || ratings.length == 0) {
            return 0;
        }

        int[] count = new int[ratings.length];
        Arrays.fill(count, 1);
        int sum = 0;
        for(int i = 1; i < ratings.length; i++) {
            if(ratings[i] > ratings[i - 1]) {
                count[i] = count[i - 1] + 1;
            }
        }

        for(int i = ratings.length - 1; i >= 1; i--) {
            sum += count[i];
            if(ratings[i - 1] > ratings[i] && count[i - 1] <= co
unt[i]) {  // second round has two conditions
                count[i-1] = count[i] + 1;
            }
        }
        sum += count[0];
        return sum;
    }
}
```

一、前言

创建型模式

创建型模式(Creational Pattern)对类的实例化过程进行了抽象，能够将软件模块中对象的创建和对象的使用分离。为了使软件的结构更加清晰，外界对于这些对象只需要知道它们共同的接口，而不清楚其具体的实现细节，使整个系统的设计更加符合单一职责原则。

创建型模式在创建什么(What)，由谁创建(Who)，何时创建(When)等方面都为软件设计者提供了尽可能大的灵活性。创建型模式隐藏了类的实例的创建细节，通过隐藏对象如何被创建和组合在一起达到使整个系统独立的目的。

包含模式

- 简单工厂模式（**Simple Factory**）
- 工厂方法模式（**Factory Method**）
- 抽象工厂模式（**Abstract Factory**）
- 建造者模式（**Builder**）
- 原型模式（**Prototype**）
- 单例模式（**Singleton**）

二、目录

本部分没有包含以上所有模式，仅介绍了几种常用的。

- 简单工厂模式
- 工厂方法模式
- 抽象工厂模式
- 单例模式
- 建造者模式

时间： 2018-05-12 01:44:07

一、简单工厂模式简介

1.定义

简单工厂模式(Simple Factory Pattern)：又称为静态工厂方法(Static Factory Method)模式，它属于类创建型模式。在简单工厂模式中，可以根据参数的不同返回不同类的实例。简单工厂模式专门定义一个类来负责创建其他类的实例，被创建的实例通常都具有共同的父类。

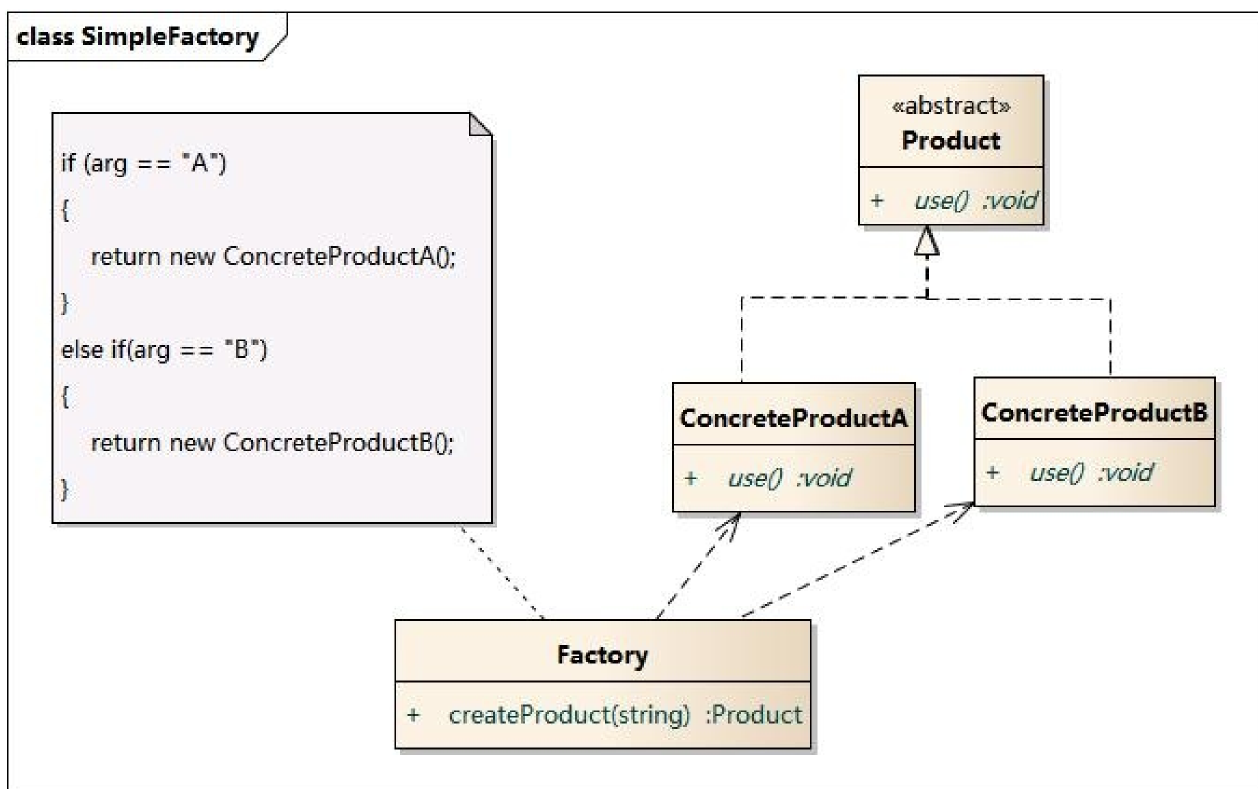
2.使用动机

考虑一个简单的软件应用场景：一个软件系统可以提供多个外观不同的按钮（如圆形按钮、矩形按钮、菱形按钮等）， 这些按钮都源自同一个基类，不过在继承基类后不同的子类修改了部分属性从而使得它们可以呈现不同的外观。

如果我们希望在使用这些按钮时，不需要知道这些具体按钮类的名字，只需要知道表示该按钮类的一个参数，并提供一个调用方便的方法，把该参数传入方法即可返回一个相应的按钮对象，此时，就可以使用简单工厂模式。

二、简单工厂模式结构

1.模式结构



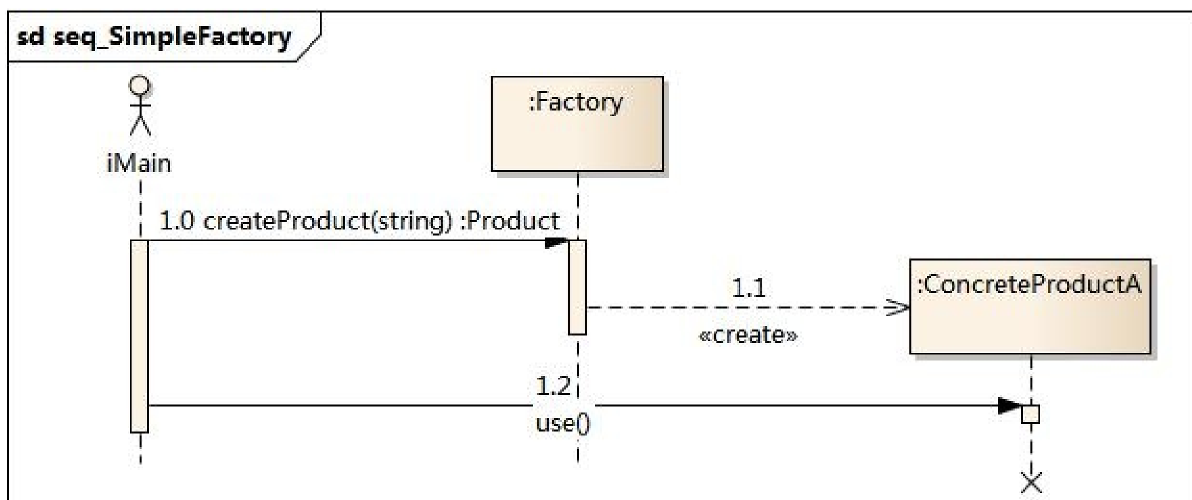
简单工厂模式包含如下角色：

●Factory：工厂角色
工厂角色负责实现创建所有实例的内部逻辑

●Product：抽象产品角色
抽象产品角色是所创建的所有对象的父类，负责描述所有实例所共有的公共接口

●ConcreteProduct：具体产品角色
具体产品角色是创建目标，所有创建的对象都充当这个角色的某个具体类的实例。

2.时序图



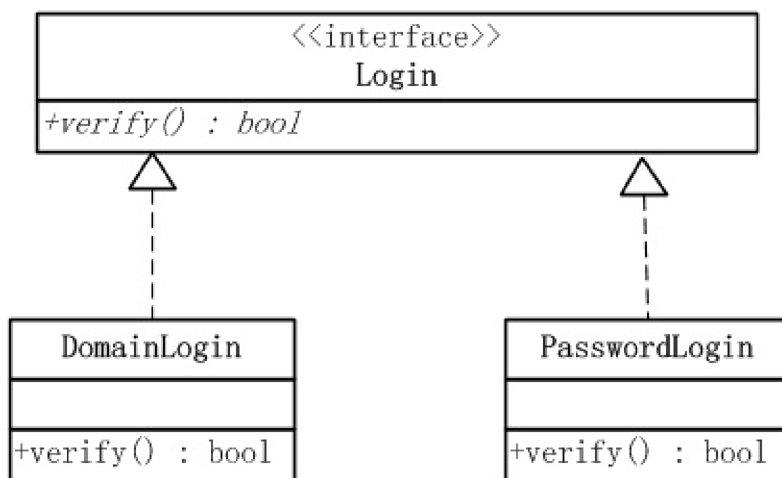
①先调用工厂类中的静态方法createProduct()

②根据传入产品类型参数，获得具体的产品对象

③返回产品对象并使用

三、简单工厂的使用实例

以登录功能来说，假如应用系统需要支持多种登录方式如：口令认证、域认证（口令认证通常是去数据库中验证用户，而域认证则是需要到微软的域中验证用户）。那么自然的做法就是建立一个各种登录方式都适用的接口，如下图所示：



抽象产品**Login**

```
public interface Login {
    //登录验证
    public boolean verify(String name , String password);
}
```

具体产品**DomainLogin**

```
public class DomainLogin implements Login {

    @Override
    public boolean verify(String name, String password) {
        // TODO Auto-generated method stub
        /**
         * 业务逻辑
         */
        return true;
    }
}
```

具体产品**PasswordLogin**

```
public class PasswordLogin implements Login {

    @Override
    public boolean verify(String name, String password) {
        // TODO Auto-generated method stub
        /**
         * 业务逻辑
         */
        return true;
    }
}
```

工厂类**LoginManager**

根据调用者不同的要求，创建出不同的登录对象并返回。而如果碰到不合法的要求，会返回一个Runtime异常。

```
public class LoginManager {
    public static Login factory(String type){
        if(type.equals("password")){

            return new PasswordLogin();

        }else if(type.equals("passcode")){

            return new DomainLogin();

        }else{
            /**
             * 这里抛出一个自定义异常会更恰当
             */
            throw new RuntimeException("没有找到登录类型");
        }
    }
}
```

测试调用

```
public class Test {
    public static void main(String[] args) {
        String loginType = "password";
        String name = "name";
        String password = "password";
        Login login = LoginManager.factory(loginType);
        boolean bool = login.verify(name, password);
        if (bool) {
            /**
             * 业务逻辑
             */
        } else {
            /**
             * 业务逻辑
             */
        }
    }
}
```

假如不使用简单工厂模式则验证登录Servlet代码如下：

```
public class Test {
    public static void main(String[] args) {
        // TODO Auto-generated method stub

        String loginType = "password";
        String name = "name";
        String password = "password";
        //处理口令认证
        if(loginType.equals("password")){
            PasswordLogin passwordLogin = new PasswordLogin();
            boolean bool = passwordLogin.verify(name, password);
            if (bool) {
                /**
                 * 业务逻辑
                 */
            } else {
                /**
                 * 业务逻辑
```

```
                 */
            }
        }
        //处理域认证
        else if(loginType.equals("passcode")){
            DomainLogin domainLogin = new DomainLogin();
            boolean bool = domainLogin.verify(name, password);
            if (bool) {
                /**
                 * 业务逻辑
                 */
            } else {
                /**
                 * 业务逻辑
                 */
            }
        }else{
            /**
             * 业务逻辑
             */
        }
    }
}
```

可以看到非常麻烦，代码重复很多，而且不利于扩展维护。

四、简单工厂模式优缺点

优点：

通过使用工厂类，外界不再需要关心如何创造各种具体的产品，只要提供一个产品的名称作为参数传给工厂，就可以直接得到一个想要的产品对象，并且可以按照接口规范来调用产品对象的所有功能（方法）。

构造容易，逻辑简单。

缺点：

1.简单工厂模式中的if else判断非常多，完全是Hard Code，如果有一个新产品要加进来，就要同时添加一个新产品类，并且必须修改工厂类，再加入一个 else if 分支才可以， 这样就违背了“开放-关闭原则”中的对修改关闭的准则了。当系统中的具体产品类不断增多时候，就要不断的修改工厂类，对系统的维护和扩展不利。

2.一个工厂类中集合了所有的类的实例创建逻辑，违反了高内聚的责任分配原则，将全部的创建逻辑都集中到了一个工厂类当中，所有的业务逻辑都在这个工厂类中实现。什么时候它不能工作了，整个系统都会受到影响。因此一般只在很简单的情况下应用，比如当工厂类负责创建的对象比较少时。

3.简单工厂模式由于使用了静态工厂方法，造成工厂角色无法形成基于继承的等级结构。

适用环境

在以下情况下可以使用简单工厂模式：

工厂类负责创建的对象比较少：由于创建的对象较少，不会造成工厂方法中的业务逻辑太过复杂。

客户端只知道传入工厂类的参数，对于如何创建对象不关心：客户端既不需要关心创建细节，甚至连类名都不需要记住，只需要知道类型所对应的参数。

五、简单工厂模式在Java中的应用

①JDK类库中广泛使用了简单工厂模式，如工具类java.text.DateFormat，它用于格式化一个本地日期或者时间。

```
public final static DateFormat getDateInstance();
public final static DateFormat getDateInstance(int style);
public final static DateFormat getDateInstance(int style,Locale
locale);
```

②Java加密技术
获取不同加密算法的密钥生成器:

```
KeyGenerator keyGen=KeyGenerator.getInstance("DESede");
```

创建密码器:

```
Cipher cp=Cipher.getInstance("DESede");
```

一、工厂方法模式简介

1.定义

工厂方法模式(Factory Method Pattern)又称为工厂模式，也叫虚拟构造器(Virtual Constructor)模式或者多态工厂(Polymorphic Factory)模式，它属于类创建型模式。

在工厂方法模式中，工厂父类负责定义创建产品对象的公共接口，而工厂子类则负责生成具体的产品对象，这样做的目的是将产品类的实例化操作延迟到工厂子类中完成，即通过工厂子类来确定究竟应该实例化哪一个具体产品类。

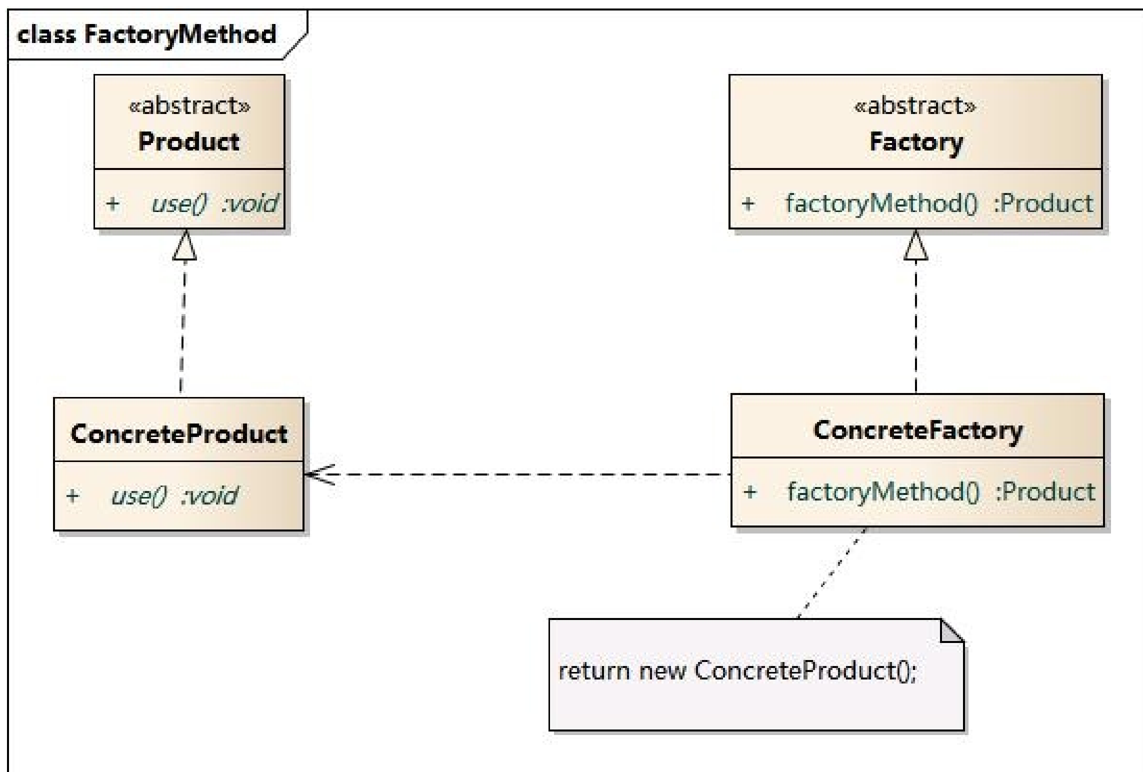
2.使用动机

现在对该系统（上篇文章提到）进行修改，不再设计一个按钮工厂类来统一负责所有产品的创建，而是将具体按钮的创建过程交给专门的工厂子类去完成。

我们先定义一个抽象的按钮工厂类，再定义具体的工厂类来生成圆形按钮、矩形按钮、菱形按钮等，它们实现在抽象按钮工厂类中定义的方法。这种抽象化的结果使这种结构可以在不修改具体工厂类的情况下引进新的产品，如果出现新的按钮类型，只需要为这种新类型的按钮创建一个具体的工厂类就可以获得该新按钮的实例，这一特点无疑使得工厂方法模式具有超越简单工厂模式的优越性，更加符合“开闭原则”。

二、工厂方法模式结构

1.模式结构



工厂方法模式包含如下角色：

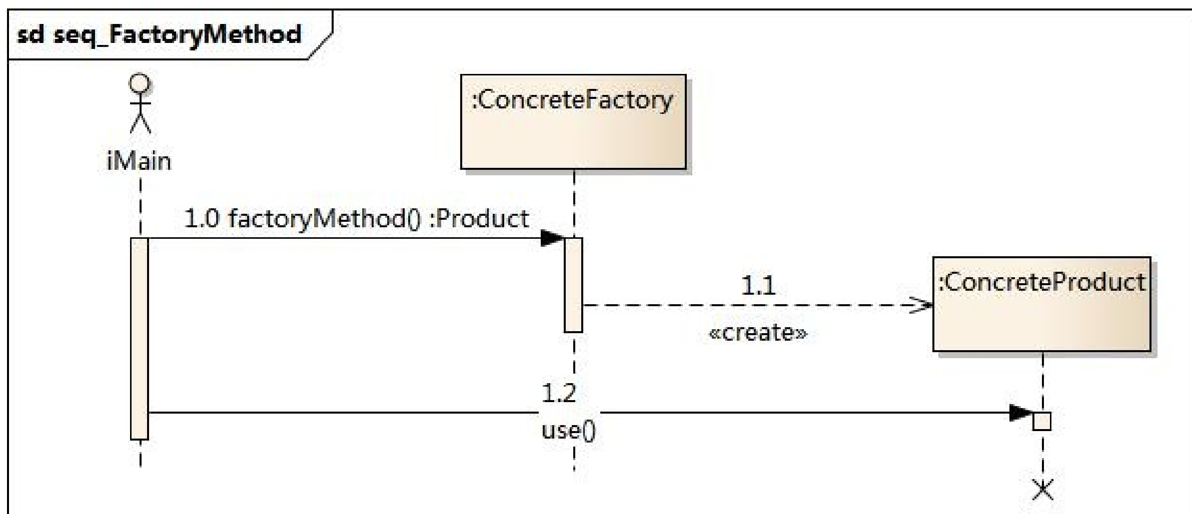
●Product：抽象产品，工厂方法模式所创建的对象的超类，也就是所有产品类的共同父类或共同拥有的接口。在实际的系统中，这个角色也常常使用抽象类实现。

●ConcreteProduct：具体产品，这个角色实现了抽象产品（Product）所声明的接口，工厂方法模式所创建的每一个对象都是某个具体产品的实例。

●Factory：抽象工厂，担任这个角色的是工厂方法模式的核心，任何在模式中创建对象的工厂类必须实现这个接口。在实际的系统中，这个角色也常常使用抽象类实现。

●ConcreteFactory：具体工厂，担任这个角色的是实现了抽象工厂接口的具体Java类。具体工厂角色含有与业务密切相关的逻辑，并且受到使用者的调用以创建具体产品对象。

2.时序图

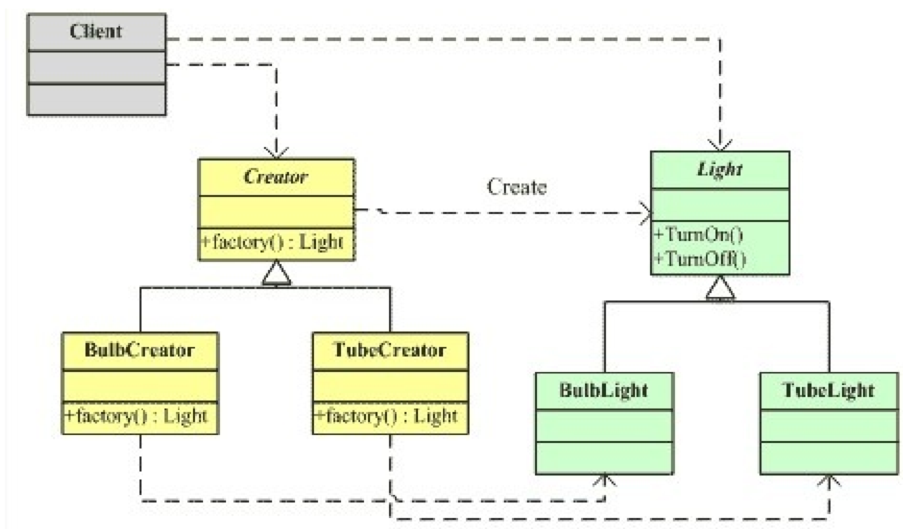


①先调用具体工厂对象中的方法createProduct()

②根据传入产品类型参数（也可以无参），获得具体的产品对象

③返回产品对象并使用

三、工厂方法模式的使用实例



上面的类图中，在灯这个品类下，有灯泡和灯管两种产品，并且都实现了灯的通用方法：关灯和开灯。在工厂类下，有各种生产具体产品的子工厂负责生产相应的两种灯具。

如果还不是太明白，那我们来假设一个情景。小明（客户端）想要买一个灯泡，他不认识工厂，只能去供销店（工厂类）买，于是和老板说“我要一个灯泡”，老板说“没问题！您稍等”。转身到了后院，对生产灯泡的小弟（灯泡工厂子类）吆喝一

声，给我造个灯泡！不一会灯泡造好了，老板拿给小明，“嘿嘿，灯泡给您作了一个，您试试？”，小明把灯泡拧在灯口上，开关了两下（灯的通用方法）“嘿！挺好，没问题！”，付了钱高高兴兴走了。

抽象的产品接口**ILight**

```
public interface ILight
{
    void TurnOn();
    void TurnOff();
}
```

具体的产品类：**BulbLight**

```
public class BulbLight implements ILight
{
    public void TurnOn()
    {
        Console.WriteLine("BulbLight turns on.");
    }
    public void TurnOff()
    {
        Console.WriteLine("BulbLight turns off.");
    }
}
```

具体的产品类：**TubeLight**

```
public class TubeLight implements ILight
{
    public void TurnOn()
    {
        Console.WriteLine("TubeLight turns on.");
    }

    public void TurnOff()
    {
        Console.WriteLine("TubeLight turns off.");
    }

}
```

抽象的工厂类

```
public interface ICreator
{
    ILight CreateLight();
}
```

具体的工厂类:**BulbCreator**

```
public class BulbCreator implements ICreator
{
    public ILight CreateLight()
    {
        return new BulbLight();
    }

}
```

具体的工厂类:**TubeCreator**

```
public class TubeCreator implements ICreator
{
    public ILight CreateLight()
    {
        return new TubeLight();
    }
}
```

客户端调用

```
static void Main(string[] args)
{
    //先给我来个灯泡
    ICreator creator = new BulbCreator();
    ILight light = creator.CreateLight();
    light.TurnOn();
    light.TurnOff();

    //再来个灯管看看
    creator = new TubeCreator();
    light = creator.CreateLight();
    light.TurnOn();
    light.TurnOff();

}
```

通过一个引用变量ICreator来创建产品对象，创建何种产品对象由指向的具体工厂类决定。通过工厂方法模式，将具体的应用逻辑和产品的创建分离开，促进松耦合。

本例中每个具体工厂类只负责生产一种类型的产品，当然每个具体工厂类也内部可以维护少数几种产品实例对象，类似于简单工厂模式。

四、工厂方法模式的优缺点

优点

①在工厂方法模式中，工厂方法用来创建客户所需要的产品，同时还向客户隐藏了哪种具体产品类将被实例化这一细节，用户只需要关心所需产品对应的工厂，无须关心创建细节，甚至无须知道具体产品类的类名。

②基于工厂角色和产品角色的多态性设计是工厂方法模式的关键。它能够使工厂可以自主确定创建何种产品对象，而如何创建这个对象的细节则完全封装在具体工厂内部。工厂方法模式之所以又被称为多态工厂模式，是因为所有的具体工厂类都具有同一抽象父类。

③使用工厂方法模式的另一个优点是在系统中加入新产品时，无须修改抽象工厂和抽象产品提供的接口，无须修改客户端，也无须修改其他的具体工厂和具体产品，而只要添加一个具体工厂和具体产品就可以了。这样，系统的可扩展性也就变得非常好，完全符合“开闭原则”，这点比简单工厂模式更优秀。

缺点

①在添加新产品时，需要编写新的具体产品类，而且还要提供与之对应的具体工厂类，系统中类的个数将成对增加，在一定程度上增加了系统的复杂度，有更多的类需要编译和运行，会给系统带来一些额外的开销。

②由于考虑到系统的可扩展性，需要引入抽象层，在客户端代码中均使用抽象层进行定义，增加了系统的抽象性和理解难度，且在实现时可能需要用到DOM、反射等技术，增加了系统的实现难度。

适用场景

在以下情况下可以使用工厂方法模式：

①一个类不知道它所需要的对象的类：在工厂方法模式中，客户端不需要知道具体产品类的类名，只需要知道所对应的工厂即可，具体的产品对象由具体工厂类创建；客户端需要知道创建具体产品的工厂类。

②一个类通过其子类来指定创建哪个对象：在工厂方法模式中，对于抽象工厂类只需要提供一个创建产品的接口，而由其子类来确定具体要创建的对象，利用面向对象的多态性和里氏代换原则，在程序运行时，子类对象将覆盖父类对象，从而使得系统更容易扩展。

③将创建对象的任务委托给多个工厂子类中的某一个，客户端在使用时可以无须关心是哪一个工厂子类创建产品子类，需要时再动态指定，可将具体工厂类的类名存储在配置文件或数据库中。

五、工厂方法模式在Java中应用

JDBC中的工厂方法:

```
Connection conn=DriverManager.getConnection("jdbc:microsoft:sqlserver://localhost:1433; DatabaseName=DB;user=sa;password=");
Statement statement=conn.createStatement();
ResultSet rs=statement.executeQuery("select * from UserInfo");
```

一、抽象工厂模式简介

1.定义

抽象工厂模式(Abstract Factory Pattern)：提供一个创建一系列相关或相互依赖对象的接口，而无须指定它们具体的类。抽象工厂模式又称为Kit模式，属于对象创建型模式。

定义很难懂？没错，看起来是很抽象，不过这正反应了这种模式的强大。下面具体阐述下定义。

2.定义阐述

在工厂方法模式中具体工厂负责生产具体的产品，每一个具体工厂对应一种或几种具体产品，工厂方法也具有唯一性，一般情况下，一个具体工厂中只有一个工厂方法或者一组重载的工厂方法。但是有时候我们需要一个工厂可以提供多个不同种类产品对象，而不是单一种类的产品对象。

为了更清晰地理解工厂方法模式，需要先引入两个概念：

产品等级结构： 产品等级结构即产品的继承结构，如一个抽象类是电视机，其子类有海尔电视机、海信电视机、TCL电视机，则抽象电视机与具体品牌的电视机之间构成了一个产品等级结构，抽象电视机是父类，而具体品牌的电视机是其子类。

产品族： 在抽象工厂模式中，产品族是指由同一个工厂生产的，位于不同产品等级结构中的一组产品，如海尔电器工厂生产的海尔电视机、海尔电冰箱，海尔电视机位于电视机产品等级结构中，海尔电冰箱位于电冰箱产品等级结构中。

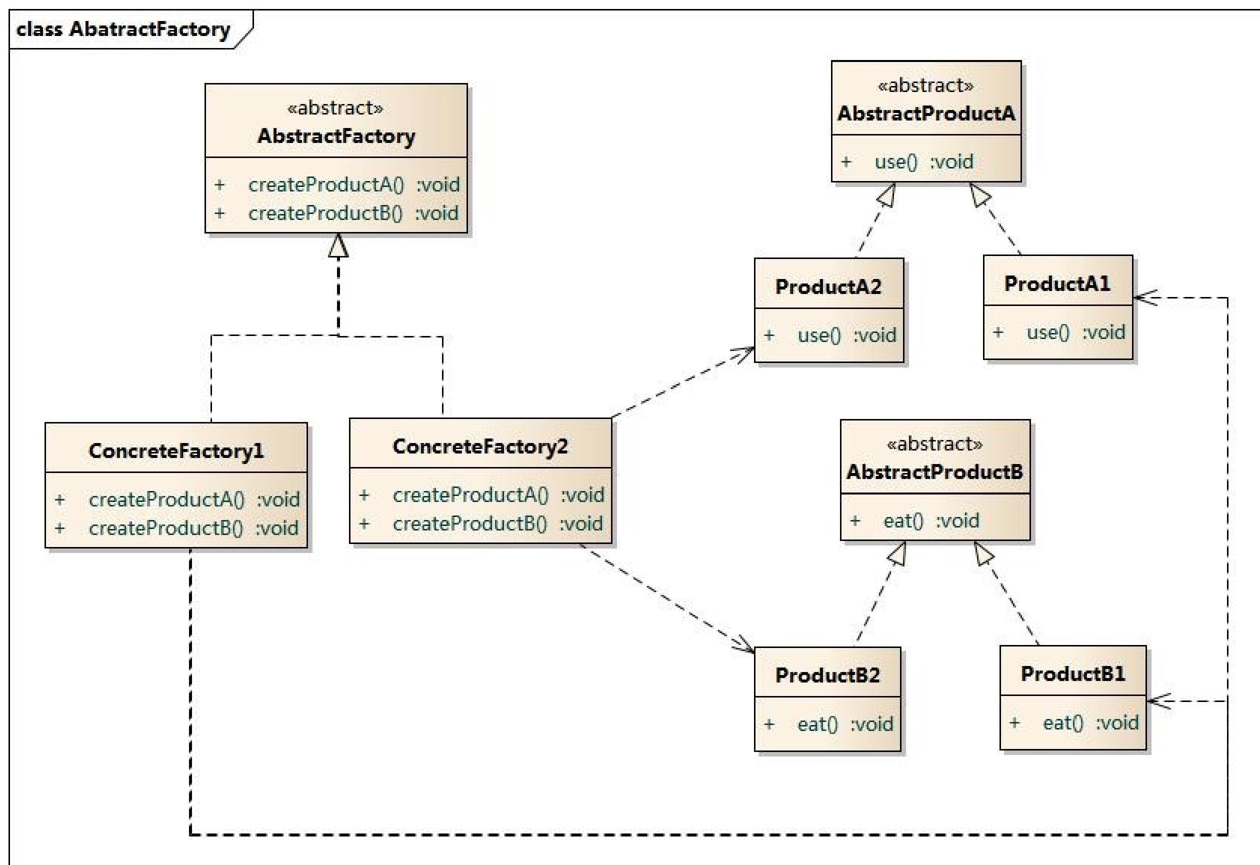
当系统所提供的工厂所需生产的具体产品并不是一个简单的对象，而是多个位于不同产品等级结构中属于不同类型的具体产品时需要使用抽象工厂模式。

抽象工厂模式是所有形式的工厂模式中最为抽象和最具一般性的一种形态。

抽象工厂模式与工厂方法模式最大的区别在于，工厂方法模式针对的是一个产品等级结构，而抽象工厂模式则需要面对多个产品等级结构，一个工厂等级结构可以负责多个不同产品等级结构中的产品对象的创建 。当一个工厂等级结构可以创建出分属于不同产品等级结构的一个产品族中的所有对象时，抽象工厂模式比工厂方法模式更为简单、有效率。

二、抽象工厂模式结构

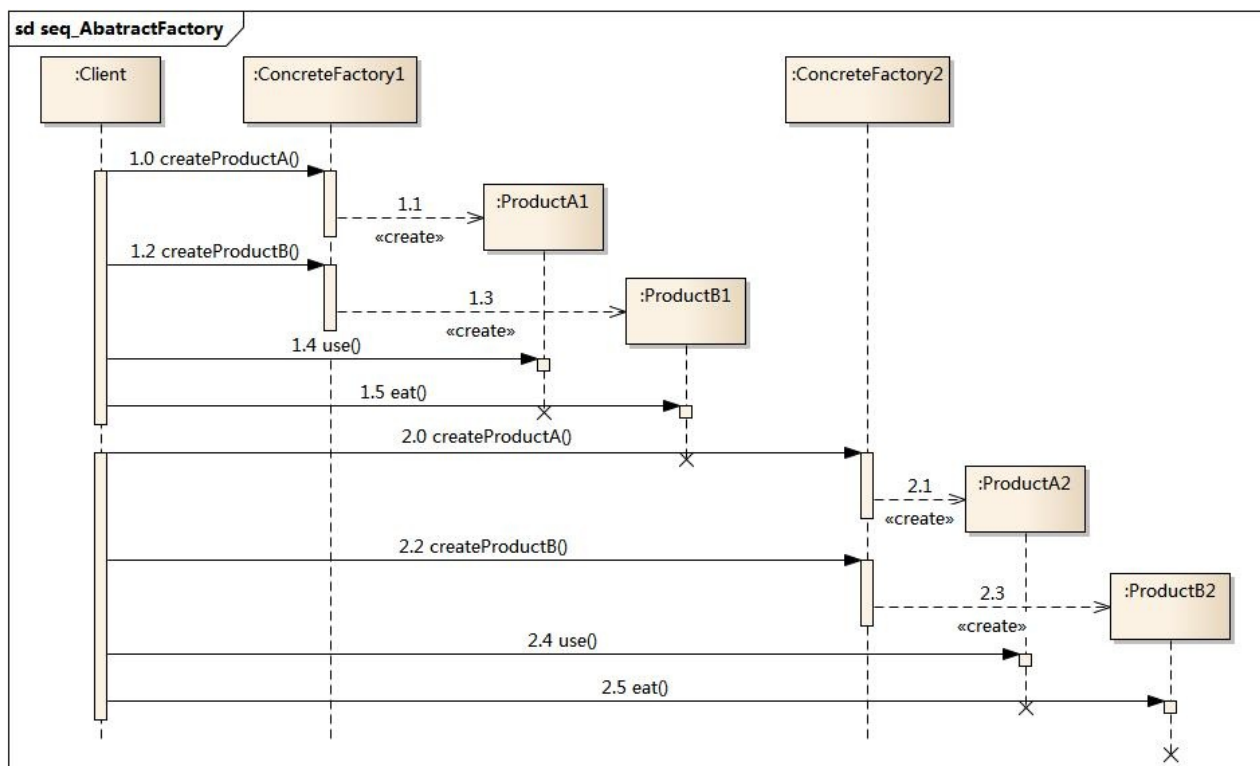
1.模式结构



抽象工厂模式包含如下角色：

AbstractFactory：抽象工厂
ConcreteFactory：具体工厂
AbstractProduct：抽象产品
Product：具体产品

2.时序图



①先调用具体工厂对象中的方法createProductX()。根据具体工厂不同可以选择不同的方法，针对同一种工厂也可以选择不同的方法创建不同类型的产品对象。

②根据传入产品类型参数（也可以无参），获得具体的产品对象

③返回产品对象并使用

三、抽象工厂的使用实例

假设有一个移动终端工厂，可以制造苹果系列的移动产品和三星系列的移动产品。这个工厂下有两个子厂，一个负责制造苹果系列的Pad和三星系列的Pad，另一个负责制造苹果系列的手机和三星系列的手机。这便是一个典型的抽象工厂的实例。
抽象产品： 苹果系列

```
public interface Apple
{
    void AppleStyle();
}
```

抽象产品： 三星系列

```
public interface Sumsung
{
    void BangziStyle();
}
```

具体产品：**iphone**

```
public class iphone implements Apple
{
    public void AppleStyle()
    {
        Console.WriteLine("Apple's style: iPhone!");
    }
}
```

具体产品：**ipad**

```
public class ipad implements Apple
{
    public void AppleStyle()
    {
        Console.WriteLine("Apple's style: iPad!");
    }
}
```

具体产品：**note2**

```
public class note2 implements Sumsung
{
    public void BangziStyle()
    {
        Console.WriteLine("Bangzi's style : Note2!");
    }
}
```

具体产品：**tabs**

```
public class Tabs implements Sumsung
{
    public void BangziStyle()
    {
        Console.WriteLine("Bangzi's style : Tab!");
    }
}
```

抽象工厂

```
public interface Factory
{
    Apple createAppleProduct();
    Sumsung createSumsungProduct();
}
```

手机工厂

```
public class Factory_Phone implements Factory
{
    public Apple createAppleProduct()
    {
        return new iphone();
    }

    public Sumsung createSumsungProduct()
    {
        return new note2();
    }
}
```

pad工厂

```
public class Factory_Pad implements  Factory
{
    public Apple createAppleProduct()
    {
        return new ipad();
    }

    public Sumsung createSumsungProduct()
    {
        return new Tabs();
    }
}
```

客户端调用

```
public static void Main(string[] args)
{
    //采购商要一台iPad和一台Tab
    Factory factory = new Factory_Pad();
    Apple apple = factory.createAppleProduct();
    apple.AppleStyle();
    Sumsung sumsung = factory.createSumsungProduct();
    sumsung.BangziStyle();

    //采购商又要一台iPhone和一台Note2
    factory = new Factory_Phone();
    apple = factory.createAppleProduct();
    apple.AppleStyle();
    sumsung = factory.createSumsungProduct();
    sumsung.BangziStyle();
    Console.ReadKey();
}
```

抽象工厂可以通过多态，来动态设置不同的工厂，生产不同的产品，同时每个工厂中的产品又不属于同一个产品等级结构。

四、抽象工厂模式优缺点

优点

①抽象工厂模式隔离了具体类的生成，使得客户并不需要知道什么被创建。由于这种隔离，更换一个具体工厂就变得相对容易。所有的具体工厂都实现了抽象工厂中定义的那些公共接口，因此只需改变具体工厂的实例，就可以在某种程度上改变整个软件系统的行为。另外，应用抽象工厂模式可以实现高内聚低耦合的设计目的，因此抽象工厂模式得到了广泛的应用。

②增加新的具体工厂和产品族很方便，因为一个具体的工厂实现代表的是一个产品族，无须修改已有系统，符合“开闭原则”。

缺点

在添加新的产品对象（不同于现有的产品等级结构）时，难以扩展抽象工厂来生产新种类的产品，这是因为在抽象工厂角色中规定了所有可能被创建的产品集合，要支持新种类的产品就意味着要对该接口进行扩展，而这将涉及到对抽象工厂角色及其所有子类的修改，显然会带来较大的不便。

开闭原则的倾斜性（增加新的工厂和产品族容易，增加新的产品等级结构麻烦）。

适用环境

在以下情况下可以使用抽象工厂模式：

①一个系统不应当依赖于产品类实例如何被创建、组合和表达的细节，这对于所有类型的工厂模式都是重要的。

②系统中有多于一个的产品族，而每次只使用其中某一产品族。与工厂方法模式的区别

③属于同一个产品族的产品将在一起使用，这一约束必须在系统的设计中体现出来。

④系统提供一个产品类的库，所有的产品以同样的接口出现，从而使客户端不依赖于具体实现。

一、单例模式

1.定义

作为对象的创建模式，单例模式确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例。这个类称为单例类。

2.特点

单例类只能有一个实例。

单例类必须自己创建自己的唯一实例。

单例类必须给所有其他对象提供这一实例。

二、创建单例模式的方式

①懒汉式，线程不安全

懒汉式其实是一种比较形象的称谓。既然懒，那么在创建对象实例的时候就不着急。会一直等到马上要使用对象实例的时候才会创建，懒人嘛，总是推脱不开的时候才会真正去执行工作，因此在装载对象的时候不创建对象实例。

```
public class Singleton {
    private static Singleton instance;
    private Singleton (){}

    public static Singleton getInstance() {
     if (instance == null) {
         instance = new Singleton();
     }
     return instance;
    }
}
```

这段代码简单明了，而且使用了懒加载模式，但是却存在致命的问题。当有多个线程并行调用 getInstance() 的时候，就会创建多个实例。也就是说在多线程下不能正常工作。

②懒汉式，线程安全

为了解决上面的问题，最简单的方法是将整个 getInstance() 方法设为同步（synchronized）。

```
public static synchronized Singleton getInstance() {
    if (instance == null) {
        instance = new Singleton();
    }
    return instance;
}
```

虽然做到了线程安全，并且解决了多实例的问题，但是它并不高效。因为在任何时候只能有一个线程调用 getInstance() 方法。但是同步操作只需要在第一次调用时才被需要，即第一次创建单例实例对象时。这就引出了双重检验锁。

③双重检验锁

双重检验锁模式（double checked locking pattern），是一种使用同步块加锁的方法。程序员称其为双重检查锁，因为会有两次检查 instance == null，一次是在同步块外，一次是在同步块内。为什么在同步块内还要再检验一次？因为可能会有多个线程一起进入同步块外的 if，如果在同步块内不进行二次检验的话就会生成多个实例了。

```
public static Singleton getSingleton() {
    if (instance == null) {                         //Single Che
cked
        synchronized (Singleton.class) {
            if (instance == null) {                 //Double Che
cked
                instance = new Singleton();
            }
        }
    }
    return instance ;
}
```

这段代码看起来很完美，很可惜，它是有问题。主要在于instance = new Singleton()这句，这并非是一个原子操作，事实上在 JVM 中这句话大概做了下面 3 件事情:

1.给 instance 分配内存

2.调用 Singleton 的构造函数来初始化成员变量

3.将instance对象指向分配的内存空间（执行完这步 instance 就为非 null 了）。

但是在 JVM 的即时编译器中存在指令重排序的优化。也就是说上面的第二步和第三步的顺序是不能保证的，最终的执行顺序可能是 1-2-3 也可能是 1-3-2。如果是后者，则在 3 执行完毕、2 未执行之前，被线程二抢占了，这时 instance 已经是非 null 了（但却没有初始化），所以线程二会直接返回 instance，然后使用，然后顺理成章地报错。

我们只需要将 instance 变量声明成 volatile 就可以了。

```
public class Singleton {
    private volatile static Singleton instance; //声明成 volatile
    private Singleton (){}

    public static Singleton getSingleton() {
        if (instance == null) {
            synchronized (Singleton.class) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }

}
```

有些人认为使用 volatile 的原因是可见性，也就是可以保证线程在本地不会存有 instance 的副本，每次都是去主内存中读取。但其实是不对的。使用 **volatile** 的主要原因是其另一个特性：禁止指令重排序优化。也就是说，在 **volatile** 变量的赋值

操作后面会有一个内存屏障（生成的汇编代码上），读操作不会被重排序到内存屏障之前。比如上面的例子，取操作必须在执行完 1-2-3 之后或者 1-3-2 之后，不存在执行到 1-3 然后取到值的情况。

从「先行发生原则」的角度理解的话，就是对于一个 volatile 变量的写操作都先行发生于后面对这个变量的读操作（这里的“后面”是时间上的先后顺序）。

但是特别注意在 Java 5 以前的版本使用了 volatile 的双检锁还是有问题的。其原因是 Java 5 以前的 JMM （Java 内存模型）是存在缺陷的，即时将变量声明成 volatile 也不能完全避免重排序，主要是 volatile 变量前后的代码仍然存在重排序问题。这个 volatile 屏蔽重排序的问题在 Java 5 中才得以修复，所以在这之后才可以放心使用 volatile。

相信你不会喜欢这种复杂又隐含问题的方式，当然我们有更好的实现线程安全的单例模式的办法。

④饿汉式 static final field

饿汉式其实是一种比较形象的称谓。既然饿，那么在创建对象实例的时候就比较着急，饿了嘛，于是在装载类的时候就创建对象实例。

这种方法非常简单，因为单例的实例被声明成 static 和 final 变量了，在第一次加载类到内存中时就会初始化，所以创建实例本身是线程安全的。

```
public class Singleton{
    //类加载时就初始化
    private static final Singleton instance = new Singleton();

    private Singleton(){}

    public static Singleton getInstance(){
        return instance;
    }
}
```

缺点是它不是一种懒加载模式（lazy initialization），单例会在加载类后一开始就被初始化，即使客户端没有调用 **getInstance()**方法。

饿汉式的创建方式在一些场景中将无法使用：譬如 Singleton 实例的创建是依赖参数或者配置文件的，在 getInstance() 之前必须调用某个方法设置参数给它，那样这种单例写法就无法使用了。

⑤静态内部类 static nested class

这种方法也是《Effective Java》上所推荐的。

```
public class Singleton {
    private static class SingletonHolder {
        private static final Singleton INSTANCE = new Singleton(
);
    }
    private Singleton (){}
    public static final Singleton getInstance() {
        return SingletonHolder.INSTANCE;
    }
}
```

这种写法仍然使用JVM本身机制保证了线程安全问题。由于静态单例对象没有作为Singleton的成员变量直接实例化，因此类加载时不会实例化Singleton，第一次调用**getInstance()**时将加载内部类**SingletonHolder**，在该内部类中定义了一个static类型的变量INSTANCE ，此时会首先初始化这个成员变量，由Java虚拟机来保证其线程安全性，确保该成员变量只能初始化一次。由于getInstance()方法没有任何线程锁定，因此其性能不会造成任何影响。

由于 SingletonHolder 是私有的，除了 getInstance() 之外没有办法访问它，因此它是懒汉式的；同时读取实例的时候不会进行同步，没有性能缺陷；也不依赖 JDK 版本。

⑥枚举 Enum

用枚举写单例实在太简单了！这也是它最大的优点。下面这段代码就是声明枚举实例的通常做法。

```
public enum EasySingleton{
    INSTANCE;
}
```

我们可以通过EasySingleton.INSTANCE来访问实例，这比调用getInstance()方法简单多了。创建枚举默认就是线程安全的，所以不需要担心double checked locking，而且还能防止反序列化导致重新创建新的对象。

三、总结

一般来说，单例模式有五种写法：懒汉、饿汉、双重检验锁、静态内部类、枚举。上述所说都是线程安全的实现，上文中第一种方式线程不安全，排除。

一般情况下直接使用饿汉式就好了，如果明确要求要懒加载（lazy initialization）倾向于使用静态内部类。如果涉及到反序列化创建对象时会试着使用枚举的方式来实现单例。

一、模式定义

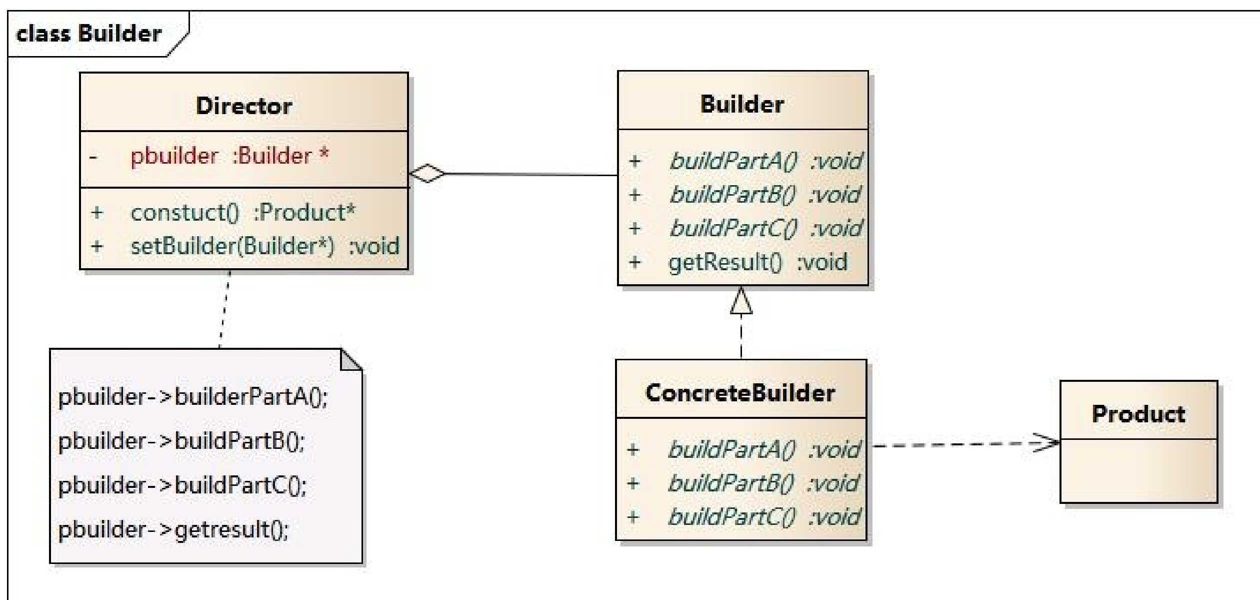
造者模式(Builder Pattern)：将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。

建造者模式是一步一步创建一个复杂的对象，它允许用户只通过指定复杂对象的类型和内容就可以构建它们，用户不需要知道内部的具体构建细节。建造者模式属于对象创建型模式。根据中文翻译的不同，建造者模式又可以称为生成器模式。

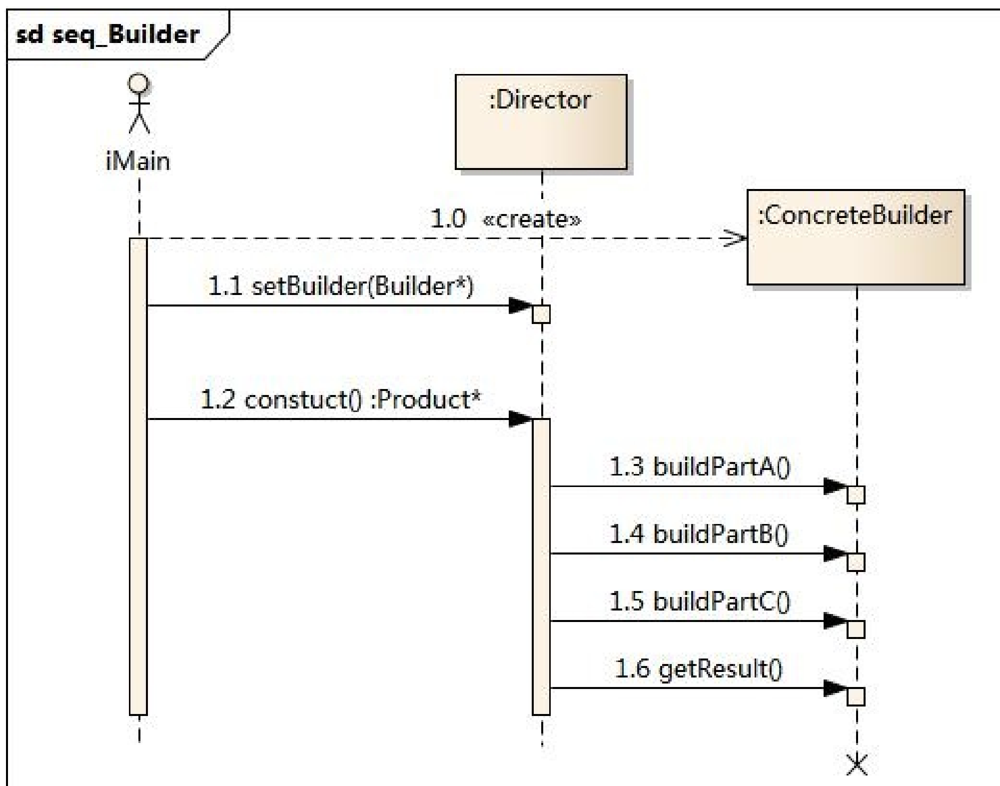
二、模式结构

建造者模式包含如下角色：

- Builder：抽象建造者
- ConcreteBuilder：具体建造者
- Director：指挥者
- Product：产品角色



三、时序图



四、简单实现

电脑的组装过程较为复杂，步骤繁多，但是顺序却是不固定的。下面我们以组装电脑为例来演示一下简单且经典的builder模式

```
package com.dp.example.builder;

/**
 * Computer产品抽象类, 为了例子简单, 只列出这几个属性
 *
 * @author mrsimple
 *
 */
public abstract class Computer {

    protected int mCpuCore = 1;
    protected int mRamSize = 0;
    protected String mOs = "Dos";

    protected Computer() {

    }
```

```
    // 设置CPU核心数
    public abstract void setCPU(int core);

    // 设置内存
    public abstract void setRAM(int gb);

    // 设置操作系统
    public abstract void setOs(String os);

    @Override
    public String toString() {
        return "Computer [mCpuCore=" + mCpuCore + ", mRamSize="
+ mRamSize
                + ", mOs=" + mOs + "]";
    }

}

package com.dp.example.builder;

/**
 * Apple电脑
 */
public class AppleComputer extends Computer {

    protected AppleComputer() {

    }

    @Override
    public void setCPU(int core) {
        mCpuCore = core;
    }

    @Override
    public void setRAM(int gb) {
        mRamSize = gb;
    }
```

```
    @Override
    public void setOs(String os) {
        mOs = os;
    }

}

package com.dp.example.builder;


package com.dp.example.builder;

/**
 * builder抽象类
 *
 */
public abstract class Builder {
    // 设置CPU核心数
    public abstract void buildCPU(int core);

    // 设置内存
    public abstract void buildRAM(int gb);

    // 设置操作系统
    public abstract void buildOs(String os);

    // 创建Computer
    public abstract Computer create();

}

package com.dp.example.builder;

public class ApplePCBuilder extends Builder {
    private Computer mApplePc = new AppleComputer();

    @Override
    public void buildCPU(int core) {
        mApplePc.setCPU(core);
    }
```

```
    @Override
    public void buildRAM(int gb) {
        mApplePc.setRAM(gb);
    }

    @Override
    public void buildOs(String os) {
        mApplePc.setOs(os);
    }

    @Override
    public Computer create() {
        return mApplePc;
    }

}

package com.dp.example.builder;

public class Director {
    Builder mBuilder = null;

    /**
     *
     * @param builder
     */
    public Director(Builder builder) {
        mBuilder = builder;
    }

    /**
     * 构建对象
     *
     * @param cpu
     * @param ram
     * @param os
     */
    public void construct(int cpu, int ram, String os) {
        mBuilder.buildCPU(cpu);
```

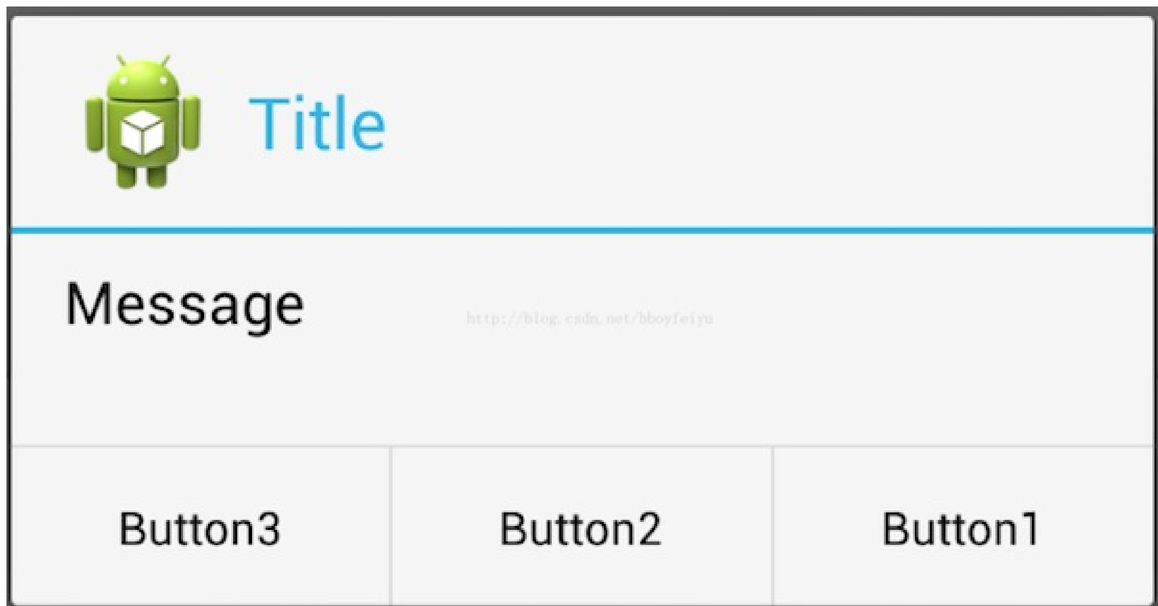
```
        mBuilder.buildRAM(ram);
        mBuilder.buildOs(os);
    }
}

/**
 * 经典实现较为繁琐
 *
 * @author mrsimple
 *
 */
public class Test {
    public static void main(String[] args) {
        // 构建器
        Builder builder = new ApplePCBuilder();
        // Director
        Director pcDirector = new Director(builder);
        // 封装构建过程, 4核, 内存2GB, Mac系统
        pcDirector.construct(4, 2, "Mac OS X 10.9.1");
        // 构建电脑, 输出相关信息
        System.out.println("Computer Info : " + builder.create()
.toString());
    }
}
```

五、Android源码中模式实现

在Android源码中，我们最常用到的Builder模式就是AlertDialog.Builder， 使用该Builder来构建复杂的AlertDialog对象。简单示例如下：

```
//显示基本的AlertDialog
private void showDialog(Context context) {
    AlertDialog.Builder builder = new AlertDialog.Builder(contex
t);
    builder.setIcon(R.drawable.icon);
    builder.setTitle("Title");
    builder.setMessage("Message");
    builder.setPositiveButton("Button1",
            new DialogInterface.OnClickListener() {
                public void onClick(DialogInterface dialog, int
whichButton) {
                    setTitle("点击了对话框上的Button1");
                }
            });
    builder.setNeutralButton("Button2",
            new DialogInterface.OnClickListener() {
                public void onClick(DialogInterface dialog, int
whichButton) {
                    setTitle("点击了对话框上的Button2");
                }
            });
    builder.setNegativeButton("Button3",
            new DialogInterface.OnClickListener() {
                public void onClick(DialogInterface dialog, int
whichButton) {
                    setTitle("点击了对话框上的Button3");
                }
            });
    builder.create().show();  // 构建AlertDialog， 并且显示
}
```

结果：

下面我们看看AlertDialog的相关源码：

```
// AlertDialog
public class AlertDialog extends Dialog implements DialogInterfa
ce {
    // Controller, 接受Builder成员变量P中的各个参数
    private AlertController mAlert;

    // 构造函数
    protected AlertDialog(Context context, int theme) {
        this(context, theme, true);
    }

    // 4 : 构造AlertDialog
    AlertDialog(Context context, int theme, boolean createContex
tWrapper) {
        super(context, resolveDialogTheme(context, theme), creat
eContextWrapper);
        mWindow.alwaysReadCloseOnTouchAttr();
        mAlert = new AlertController(getContext(), this, getWind
ow());
    }

    // 实际上调用的是mAlert的setTitle方法
    @Override
    public void setTitle(CharSequence title) {
```

```
        super.setTitle(title);
        mAlert.setTitle(title);
    }

    // 实际上调用的是mAlert的setCustomTitle方法
    public void setCustomTitle(View customTitleView) {
        mAlert.setCustomTitle(customTitleView);
    }

    public void setMessage(CharSequence message) {
        mAlert.setMessage(message);
    }

    // AlertDialog其他的代码省略

    // ************  Builder为AlertDialog的内部类   **************
*****
    public static class Builder {
        // 1 : 存储AlertDialog的各个参数, 例如title, message, icon
等.
        private final AlertController.AlertParams P;
        // 属性省略

        /**
         * Constructor using a context for this builder and the {
@link AlertDialog} it creates.
         */
        public Builder(Context context) {
            this(context, resolveDialogTheme(context, 0));
        }


        public Builder(Context context, int theme) {
            P = new AlertController.AlertParams(new ContextTheme
Wrapper(
                    context, resolveDialogTheme(context, theme))
);
            mTheme = theme;
        }
```

```
        // Builder的其他代码省略 ......

        // 2 : 设置各种参数
        public Builder setTitle(CharSequence title) {
            P.mTitle = title;
            return this;
        }


        public Builder setMessage(CharSequence message) {
            P.mMessage = message;
            return this;
        }

        public Builder setIcon(int iconId) {
            P.mIconId = iconId;
            return this;
        }

        public Builder setPositiveButton(CharSequence text, final
 OnClickListener listener) {
            P.mPositiveButtonText = text;
            P.mPositiveButtonListener = listener;
            return this;
        }


        public Builder setView(View view) {
            P.mView = view;
            P.mViewSpacingSpecified = false;
            return this;
        }

        // 3 : 构建AlertDialog, 传递参数
        public AlertDialog create() {
            // 调用new AlertDialog构造对象， 并且将参数传递个体AlertD
ialog
            final AlertDialog dialog = new AlertDialog(P.mContex
t, mTheme, false);
            // 5 : 将P中的参数应用的dialog中的mAlert对象中
```

```
                P.apply(dialog.mAlert);
                dialog.setCancelable(P.mCancelable);
                if (P.mCancelable) {
                    dialog.setCanceledOnTouchOutside(true);
                }
                dialog.setOnCancelListener(P.mOnCancelListener);
                if (P.mOnKeyListener != null) {
                    dialog.setOnKeyListener(P.mOnKeyListener);
                }
                return dialog;
            }
        }

    }
```

可以看到，通过Builder来设置AlertDialog中的title, message, button等参数， 这些参数都存储在类型为AlertController.AlertParams的成员变量P中，AlertController.AlertParams中包含了与之对应的成员变量。在调用Builder类的create函数时才创建AlertDialog, 并且将Builder成员变量P中保存的参数应用到AlertDialog的mAlert对象中，即P.apply(dialog.mAlert)代码段。我们看看apply函数的实现：

```
public void apply(AlertController dialog) {
    if (mCustomTitleView != null) {
        dialog.setCustomTitle(mCustomTitleView);
    } else {
        if (mTitle != null) {
            dialog.setTitle(mTitle);
        }
        if (mIcon != null) {
            dialog.setIcon(mIcon);
        }
        if (mIconId >= 0) {
            dialog.setIcon(mIconId);
        }
        if (mIconAttrId > 0) {
            dialog.setIcon(dialog.getIconAttributeResId(mIconAtt
rId));
```

```
        }
    }
    if (mMessage != null) {
        dialog.setMessage(mMessage);
    }
    if (mPositiveButtonText != null) {
        dialog.setButton(DialogInterface.BUTTON_POSITIVE, mPosit
iveButtonText,
                mPositiveButtonListener, null);
    }
    if (mNegativeButtonText != null) {
        dialog.setButton(DialogInterface.BUTTON_NEGATIVE, mNegat
iveButtonText,
                mNegativeButtonListener, null);
    }
    if (mNeutralButtonText != null) {
        dialog.setButton(DialogInterface.BUTTON_NEUTRAL, mNeutra
lButtonText,
                mNeutralButtonListener, null);
    }
    if (mForceInverseBackground) {
        dialog.setInverseBackgroundForced(true);
    }
    // For a list, the client can either supply an array of item
s or an
    // adapter or a cursor
    if ((mItems != null) || (mCursor != null) || (mAdapter != nu
ll)) {
        createListView(dialog);
    }
    if (mView != null) {
        if (mViewSpacingSpecified) {
            dialog.setView(mView, mViewSpacingLeft, mViewSpacing
Top, mViewSpacingRight,
                    mViewSpacingBottom);
        } else {
            dialog.setView(mView);
        }
    }
}
```

实际上就是把P中的参数挨个的设置到AlertController中， 也就是AlertDialog中的mAlert对象。从AlertDialog的各个setter方法中我们也可以看到，实际上也都是调用了mAlert对应的setter方法。在这里，Builder同时扮演了上文中提到的builder、ConcreteBuilder、Director的角色，简化了Builder模式的设计。

六、优缺点

优点

- 良好的封装性， 使用建造者模式可以使客户端不必知道产品内部组成的细节；
- 建造者独立，容易扩展；
- 在对象创建过程中会使用到系统中的一些其它对象，这些对象在产品对象的创建过程中不易得到。

缺点

- 会产生多余的Builder对象以及Director对象，消耗内存；
- 对象的构建过程暴露。

一、前言

结构型模式

结构型模式(Structural Pattern)描述如何将类或者对 象结合在一起形成更大的结构，就像搭积木，可以通过 简单积木的组合形成复杂的、功能更为强大的结构。

结构型模式可以分为类结构型模式和对象结构型模式：

- 类结构型模式关心类的组合，由多个类可以组合成一个更大的系统，在类结构型模式中一般只存在继承关系和实现关系。
- 对象结构型模式关心类与对象的组合，通过关联关系使得在一 个类中定义另一个类的实例对象，然后通过该对象调用其方法。 根据“合成复用原则”，在系统中尽量使用关联关系来替代继 承关系，因此大部分结构型模式都是对象结构型模式。

包含模式

- 适配器模式**(Adapter)**
- 桥接模式**(Bridge)**
- 组合模式**(Composite)**
- 装饰模式**(Decorator)**
- 外观模式**(Facade)**
- 享元模式**(Flyweight)**
- 代理模式**(Proxy)**

二、目录

本部分没有包含以上所有模式，仅介绍了几种常用的。

- 适配器模式
- 外观模式
- 装饰者模式
- 代理模式

一、适配器模式简介

1.定义

适配器模式把一个类的接口变换成客户端所期待的另一种接口，从而使原本因接口不匹配而无法在一起工作的两个类能够在一起工作。

2.定义阐述

适配器提供客户类需要的接口，适配器的实现就是把客户类的请求转化为对适配者的相应接口的调用。也就是说：当客户类调用适配器的方法时，在适配器类的内部将调用适配者类的方法，而这个过程对客户类是透明的，客户类并不直接访问适配者类。因此，适配器可以使由于接口不兼容而不能交互的类可以一起工作。这就是适配器模式的模式动机。

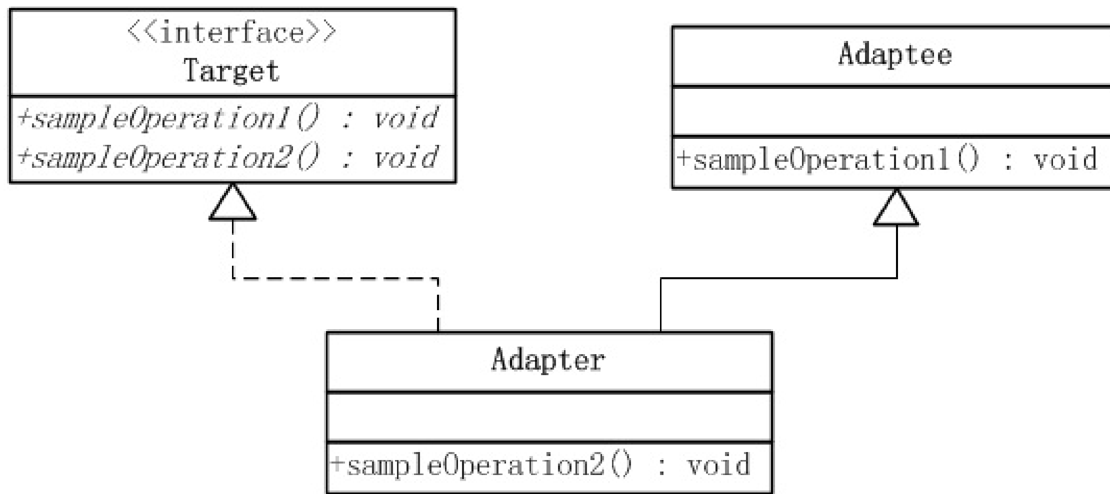
例如： 用电器做例子，笔记本电脑的插头一般都是三相的，即除了阳极、阴极外，还有一个地极。而有些地方的电源插座却只有两极，没有地极。电源插座与笔记本电脑的电源插头不匹配使得笔记本电脑无法使用。这时候一个三相到两相的转换器（适配器）就能解决此问题，而这正像是本模式所做的事情。

二、适配器模式结构

适配器模式有类的适配器模式和对象的适配器模式两种不同的形式。

类适配器模式

类的适配器模式把适配的类的API转换成为目标类的API。



在上图中可以看出，Adaptee类并没有sampleOperation2()方法，而客户端则期待这个方法。为使客户端能够使用Adaptee类，提供一个中间环节，即类Adapter，把Adaptee的API与Target类的API衔接起来。Adapter与Adaptee是继承关系，这决定了这个适配器模式是类的：

模式所涉及的角色有：

- 目标(Target)角色：这就是所期待得到的接口。注意：由于这里讨论的是类适配器模式，因此目标不可以是类。

- 源(Adapee)角色：现在需要适配的接口。

- 适配器(Adaper)角色：适配器类是本模式的核心。适配器把源接口转换成目标接口。显然，这一角色不可以是接口，而必须是具体类。

```
public interface Target {
    /**
     * 这是源类Adaptee也有的方法
     */
    public void sampleOperation1();
    /**
     * 这是源类Adapteee没有的方法
     */
    public void sampleOperation2();
}
```

上面给出的是目标角色的源代码，这个角色是以一个JAVA接口的形式实现的。可以看出，这个接口声明了两个方法：sampleOperation1()和sampleOperation2()。而源角色Adaptee是一个具体类，它有一个sampleOperation1()方法，但是没有sampleOperation2()方法。

```
public class Adaptee {

    public void sampleOperation1(){}

}
```

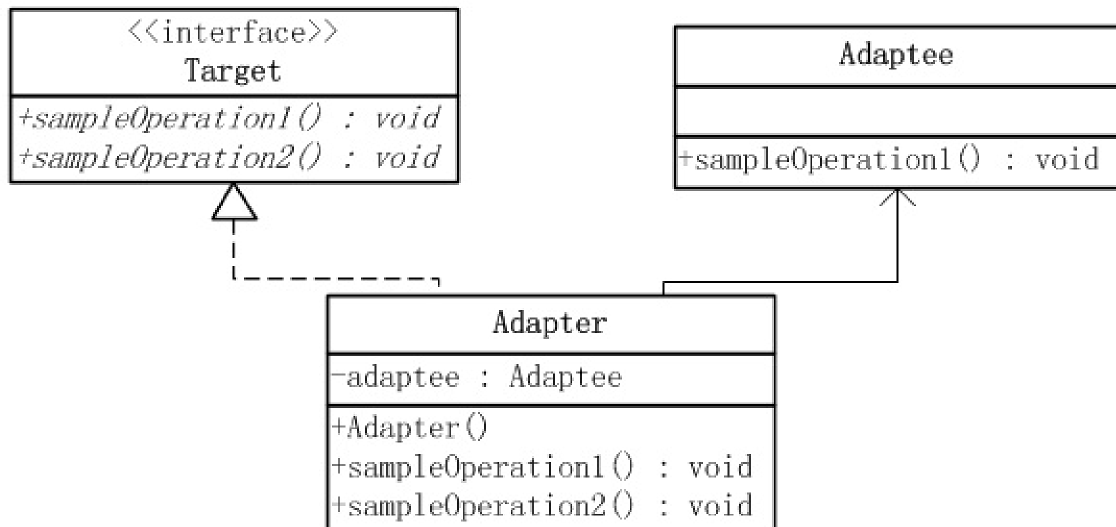
适配器角色Adapter扩展了Adaptee,同时又实现了目标(Target)接口。由于Adaptee没有提供sampleOperation2()方法，而目标接口又要求这个方法，因此适配器角色Adapter实现了这个方法。

```
public class Adapter extends Adaptee implements Target {
    /**
     * 由于源类Adaptee没有方法sampleOperation2()
     * 因此适配器补充上这个方法
     */
    @Override
    public void sampleOperation2() {
        //写相关的代码
    }

}
```

对象适配器模式

与类的适配器模式一样，对象的适配器模式把被适配的类的API转换成为目标类的API，与类的适配器模式不同的是，对象的适配器模式不是使用继承关系连接到Adaptee类，而是使用委派关系连接到Adaptee类。



从上图可以看出，Adaptee类并没有sampleOperation2()方法，而客户端则期待这个方法。为使客户端能够使用Adaptee类，需要提供一个包装(Wrapper)类Adapter。这个包装类包装了一个Adaptee的实例，从而此包装类能够把Adaptee的API与Target类的API衔接起来。Adapter与Adaptee是委派关系，这决定了适配器模式是对象的。

```
public interface Target {
    /**
     * 这是源类Adaptee也有的方法
     */
    public void sampleOperation1();
    /**
     * 这是源类Adapteee没有的方法
     */
    public void sampleOperation2();
}
```

```
public class Adaptee {

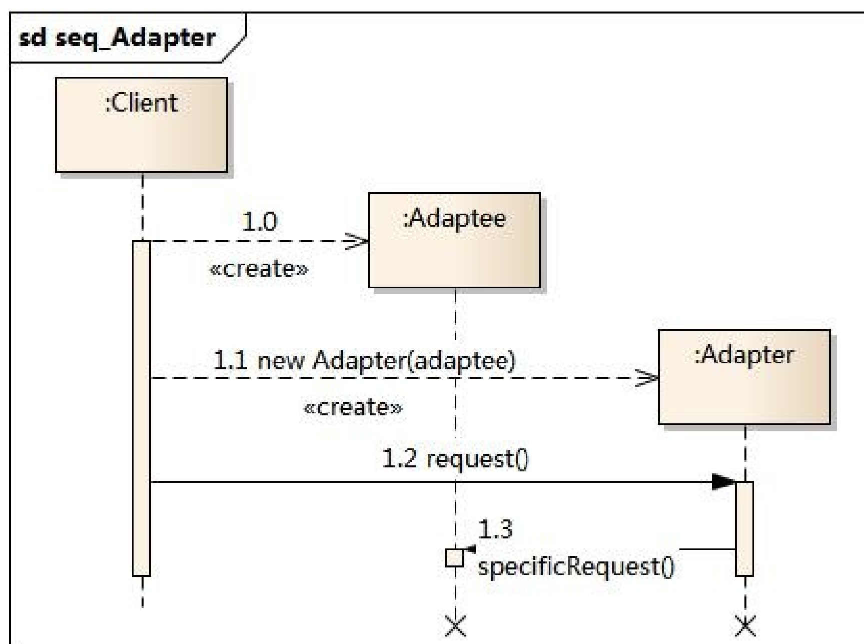
    public void sampleOperation1(){}

}
```

```
public class Adapter {
    private Adaptee adaptee;

    public Adapter(Adaptee adaptee){
        this.adaptee = adaptee;
    }
    /**
     * 源类Adaptee有方法sampleOperation1
     * 因此适配器类直接委派即可
     */
    public void sampleOperation1(){
        this.adaptee.sampleOperation1();
    }
    /**
     * 源类Adaptee没有方法sampleOperation2
     * 因此由适配器类需要补充此方法
     */
    public void sampleOperation2(){
        //写相关的代码
    }
}
```

时序图



三、类适配器和对象适配器的权衡

- 类适配器使用对象继承的方式，是静态的定义方式；而对象适配器使用对象组合的方式，是动态组合的方式。

- 对于类适配器，由于适配器直接继承了Adaptee，使得适配器不能和Adaptee的子类一起工作，因为继承是静态的关系，当适配器继承了Adaptee后，就不可能再去处理 Adaptee的子类了。

- 对于对象适配器，一个适配器可以把多种不同的源适配到同一个目标。换言之，同一个适配器可以把源类和它的子类都适配到目标接口。因为对象适配器采用的是对象组合的关系，只要对象类型正确，是不是子类都无所谓。

- 对于类适配器，适配器可以重定义Adaptee的部分行为，相当于子类覆盖父类的部分实现方法。

- 对于对象适配器，要重定义Adaptee的行为比较困难，这种情况下，需要定义Adaptee的子类来实现重定义，然后让适配器组合子类。虽然重定义Adaptee的行为比较困难，但是想要增加一些新的行为则方便的很，而且新增加的行为可同时适用于所有的源。

- 对于类适配器，仅仅引入了一个对象，并不需要额外的引用来间接得到Adaptee。

- 对于对象适配器，需要额外的引用来间接得到Adaptee。

建议尽量使用对象适配器的实现方式，多用合成/聚合、少用继承。当然，具体问题具体分析，根据需要来选用实现方式，最适合的才是最好的。

四、缺省适配器

缺省适配(Default Adapter)模式为一个接口提供缺省实现，这样子类型可以从这个缺省实现进行扩展，而不必从原有接口进行扩展。

当不需要全部实现接口提供的方法时，可先设计一个抽象类实现接口，并为该接口中每个方法提供一个默认实现（空方法），那么该抽象类的子类可有选择地覆盖父类的某些方法来实现需求，它适用于一个接口不想使用其所有的方法的情况。

五、Java中适配器模式的使用

JDK1.1 之前提供的容器有 Arrays,Vector,Stack,Hashtable,Properties,BitSet，其中定义了一种访问群集内各元素的标准方式，称为 Enumeration（列举器）接口。

```
Vector v=new Vector();
for (Enumeration enum =v.elements(); enum.hasMoreElements();) {
  Object o = enum.nextElement();
  processObject(o);
}
```

JDK1.2 版本中引入了 Iterator 接口，新版本的集合对（HashSet,HashMap,WeakHeahMap,ArrayList,TreeSet,TreeMap, LinkedList）是通过 Iterator 接口访问集合元素。

```
List list=new ArrayList();
for(Iterator it=list.iterator();it.hasNext();){
   System.out.println(it.next());
}
```

这样，如果将老版本的程序运行在新的 Java 编译器上就会出错。因为 List 接口中已经没有 elements()，而只有 iterator() 了。那么如何将老版本的程序运行在新的 Java 编译器上呢? 如果不加修改，是肯定不行的，但是修改要遵循“开－闭”原则。我们可以用 Java 设计模式中的适配器模式解决这个问题。

```
public class NewEnumeration implements Enumeration {
    Iterator it;

    public NewEnumeration(Iterator it) {
        this.it = it;
    }

    public boolean hasMoreElements() {
        return it.hasNext();
    }

    public Object nextElement() {
        return it.next();
    }

    public static void main(String[] args) {
        List list = new ArrayList();
        list.add("a");
        list.add("b");
        list.add("C");
        for (Enumeration e = new NewEnumeration(list.iterator())
; e.hasMoreElements(); ) {
            System.out.println(e.nextElement());
        }
    }
}
```

NewEnumeration 是一个适配器类，通过它实现了从 Iterator 接口到 Enumeration 接口的适配，这样我们就可以使用老版本的代码来使用新的集合对象了。

六、Android中适配器模式的使用

在开发过程中,ListView的Adapter是我们最为常见的类型之一。一般的用法大致如下:

```
 // 代码省略
 ListView myListView = (ListView) findViewById(listview_id);
// 设置适配器
```

```
 myListView.setAdapter(new MyAdapter(context,myDatas));

// 适配器
public class MyAdapter extends BaseAdapter {

    private LayoutInflater mInflater;
    List<String> mDatas;

    public MyAdapter(Context context, List<String> datas) {
        this.mInflater = LayoutInflater.from(context);
        mDatas = datas;
    }

    @Override
    public int getCount() {
        return mDatas.size();
    }

    @Override
    public String getItem(int pos) {
        return mDatas.get(pos);
    }

    @Override
    public long getItemId(int pos) {
        return pos;
    }

    // 解析、设置、缓存convertView以及相关内容
    @Override
    public View getView(int position, View convertView, ViewGrou
p parent) {
        ViewHolder holder = null;
        // Item View的复用
        if (convertView == null) {
            holder = new ViewHolder();
            convertView = mInflater.inflate(R.layout.my_listview
_item, null);
            // 获取title
            holder.title = (TextView) convertView.findViewById(R
```

```
.id.title);
                convertView.setTag(holder);
            } else {
                holder = (ViewHolder) convertView.getTag();
            }
            holder.title.setText(mDatas.get(position));
            return convertView;
        }

}
```

我们知道，作为最重要的View，ListView需要能够显示各式各样的视图，每个人需要的显示效果各不相同，显示的数据类型、数量等也千变万化。那么如何隔离这种变化尤为重要。

Android的做法是增加一个Adapter层来应对变化，将**ListView**需要的接口抽象到**Adapter**对象中，这样只要用户实现了**Adapter**的接口，**ListView**就可以按照用户设定的显示效果、数量、数据来显示特定的**Item View**。

通过代理数据集来告知ListView数据的个数(getCount函数)以及每个数据的类型(getItem函数)，最重要的是要解决Item View的输出。Item View千变万化，但终究它都是View类型，Adapter统一将Item View输出为View (getView函数)，这样就很好的应对了Item View的可变性。

那么ListView是如何通过Adapter模式 (不止Adapter模式)来运作的呢 ？我们一起来看一看。
ListView继承自AbsListView，Adapter定义在AbsListView中，我们看一看这个类。

```
public abstract class AbsListView extends AdapterView<ListAdapter
>
        implements TextWatcher,
        ViewTreeObserver.OnGlobalLayoutListener, Filter.FilterLi
stener,
        ViewTreeObserver.OnTouchModeChangeListener,
        RemoteViewsAdapter.RemoteAdapterConnectionCallback {

    ListAdapter mAdapter;

    // 关联到Window时调用的函数
```

```
    @Override
    protected void onAttachedToWindow() {
        super.onAttachedToWindow();
        // 代码省略
        // 给适配器注册一个观察者。
        if (mAdapter != null&&
        mDataSetObserver == null){
            mDataSetObserver = new AdapterDataSetObserver();
            mAdapter.registerDataSetObserver(mDataSetObserver);

            // Data may have changed while we were detached. Ref
resh.
            mDataChanged = true;
            mOldItemCount = mItemCount
            // 获取Item的数量,调用的是mAdapter的getCount方法
            mItemCount = mAdapter.getCount();
        }
        mIsAttached = true;
    }

    /**
     * 子类需要覆写layoutChildren()函数来布局child view,也就是Item Vi
ew
     */
    @Override
    protected void onLayout(boolean changed, int l, int t, int r
, int b) {
        super.onLayout(changed, l, t, r, b);
        mInLayout = true;
        if (changed) {
            int childCount = getChildCount();
            for (int i = 0; i<childCount; i++) {
                getChildAt(i).forceLayout();
            }
            mRecycler.markChildrenDirty();
        }

        if (mFastScroller != null&&
        mItemCount != mOldItemCount){
            mFastScroller.onItemCountChanged(mOldItemCount, mIte
```

```
mCount);
        }
        // 布局Child View
        layoutChildren();
        mInLayout = false;

        mOverscrollMax = (b - t) / OVERSCROLL_LIMIT_DIVISOR;
    }

    // 获取一个Item View
    View obtainView(int position, boolean[] isScrap) {
        isScrap[0] = false;
        View scrapView;
        // 从缓存的Item View中获取,ListView的复用机制就在这里
        scrapView = mRecycler.getScrapView(position);

        View child;
        if (scrapView != null) {
            // 代码省略
            child = mAdapter.getView(position, scrapView, this);
            // 代码省略
        } else {
            child = mAdapter.getView(position, null, this);
            // 代码省略
        }

        return child;
    }
}
```

通过增加Adapter一层来将Item View的操作抽象起来，ListView等集合视图通过Adapter对象获得Item的个数、数据元素、Item View等，从而达到适配各种数据、各种Item视图的效果。

因为Item View和数据类型千变万化，Android的架构师们将这些变化的部分交给用户来处理，通过getCount、getItem、getView等几个方法抽象出来，也就是将Item View的构造过程交给用户来处理，灵活地运用了适配器模式，达到了无限适配、拥抱变化的目的。

七、适配器模式的优缺点

适配器模式的优点

更好的复用性

系统需要使用现有的类，而此类的接口不符合系统的需要。那么通过适配器模式就可以让这些功能得到更好的复用。

更好的扩展性

在实现适配器功能的时候，可以调用自己开发的功能，从而自然地扩展系统的功能。

适配器模式的缺点

过多的使用适配器，会让系统非常零乱，不易整体进行把握。比如，明明看到调用的是A接口，其实内部被适配成了B接口的实现，一个系统如果太多出现这种情况，无异于一场灾难。因此如果不是很有必要，可以不使用适配器，而是直接对系统进行重构。

一、外观模式概述

1.定义

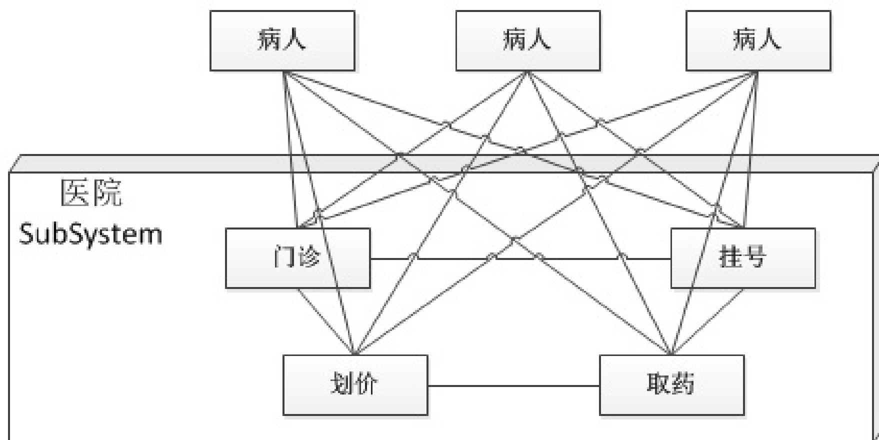
外观模式(Facade Pattern)：外部与一个子系统的通信必须通过一个统一的外观对象进行，为子系统中的一组接口提供一个一致的界面，外观模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。外观模式又称为门面模式，它是一种对象结构型模式。

2.定义阐述

医院的例子

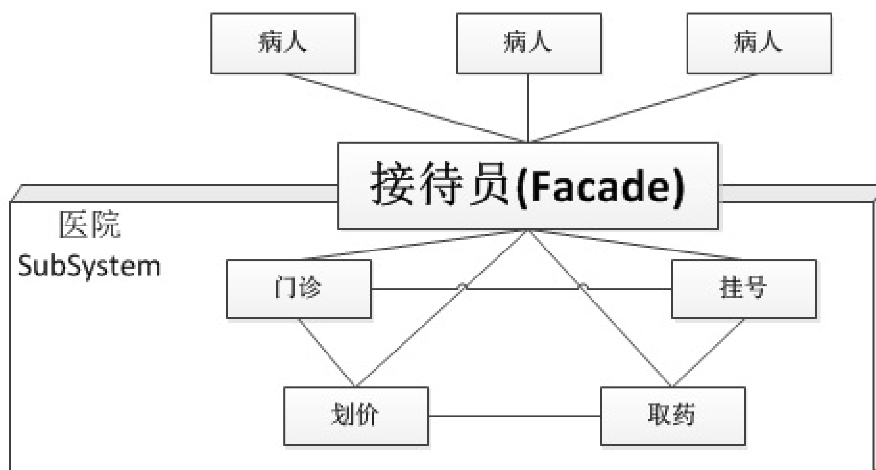
现代的软件系统都是比较复杂的，设计师处理复杂系统的一个常见方法便是将其“分而治之”，把一个系统划分为几个较小的子系统。如果把医院作为一个子系统，按照部门职能，这个系统可以划分为挂号、门诊、划价、化验、收费、取药等。看病的病人要与这些部门打交道，就如同一个子系统的客户端与一个子系统的各个类打交道一样，不是一件容易的事情。

首先病人必须先挂号，然后门诊。如果医生要求化验，病人必须首先划价，然后缴费，才可以到化验部门做化验。化验后再回到门诊室。



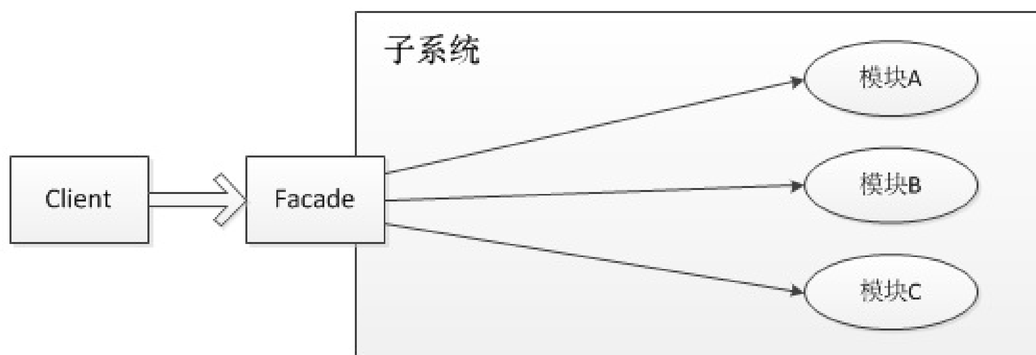
上图描述的是病人在医院里的体验，图中的方框代表医院。

解决这种不便的方法便是引进外观模式，医院可以设置一个接待员的位置，由接待员负责代为挂号、划价、缴费、取药等。这个接待员就是外观模式的体现，病人只接触接待员，由接待员与各个部门打交道。

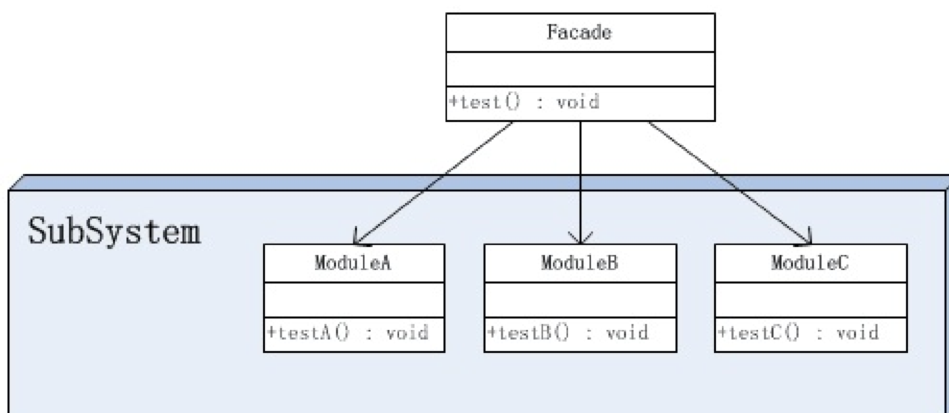


二、外观模式结构

外观模式没有一个一般化的类图描述，最好的描述方法实际上就是以一个例子说明。



由于门面模式的结构图过于抽象，因此把它稍稍具体点。假设子系统内有三个模块，分别是ModuleA、ModuleB和ModuleC，它们分别有一个示例方法，那么此时示例的整体结构图如下：

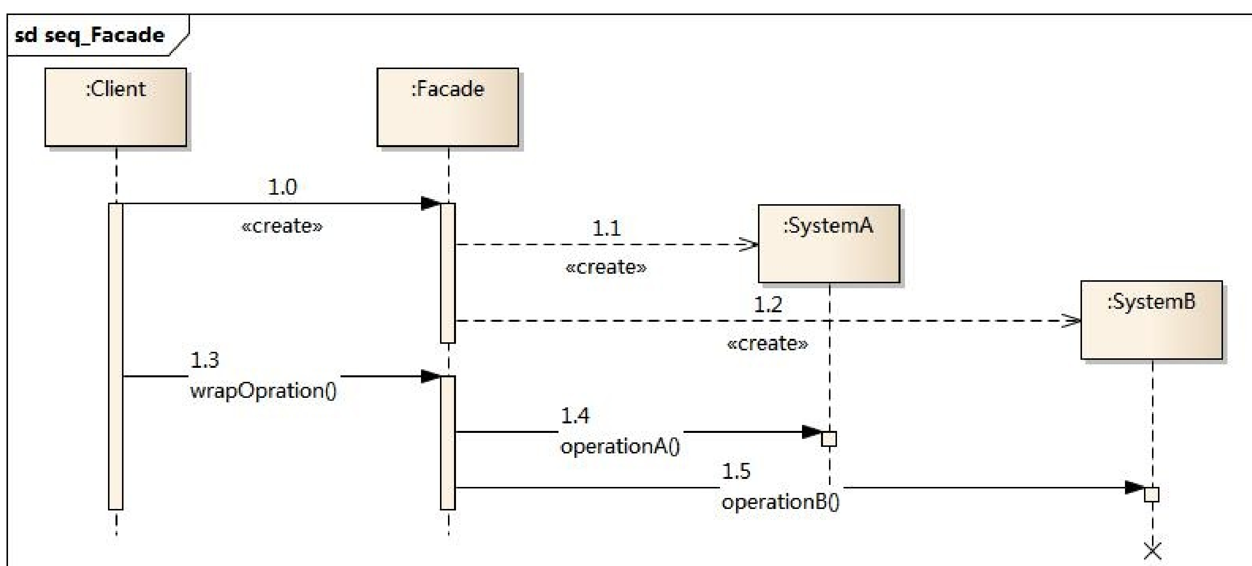


在这个对象图中，出现了两个角色：

● 外观(**Facade**)角色 ：客户端可以调用这个角色的方法。此角色知晓相关的（一个或者多个）子系统的功能和责任。在正常情况下，本角色会将所有从客户端发来的请求委派到相应的子系统去。

● 子系统(**SubSystem**)角色 ：可以同时有一个或者多个子系统。每个子系统都不是一个单独的类，而是一个类的集合（如上面的子系统就是由ModuleA、ModuleB、ModuleC三个类组合而成）。每个子系统都可以被客户端直接调用，或者被门面角色调用。子系统并不知道门面的存在，对于子系统而言，门面仅仅是另外一个客户端而已。

时序图



子系统角色中的类：

```
public class ModuleA {
    //示意方法
    public void testA(){
        System.out.println("调用ModuleA中的testA方法");
    }
}
```

```java
public class ModuleB {
    //示意方法
    public void testB(){
        System.out.println("调用ModuleB中的testB方法");
    }
}
```

```java
public class ModuleC {
    //示意方法
    public void testC(){
        System.out.println("调用ModuleC中的testC方法");
    }
}
```

外观角色类:

```java
public class Facade {
    //示意方法，满足客户端需要的功能
    public void test(){
        ModuleA a = new ModuleA();
        a.testA();
        ModuleB b = new ModuleB();
        b.testB();
        ModuleC c = new ModuleC();
        c.testC();
    }
}
```

客户端角色类：

```
public class Client {

    public static void main(String[] args) {

        Facade facade = new Facade();
        facade.test();
    }

}
```

Facade类其实相当于A、B、C模块的外观界面，有了这个Facade类，那么客户端就不需要亲自调用子系统中的A、B、C模块了，也不需要知道系统内部的实现细节，甚至都不需要知道A、B、C模块的存在，客户端只需要跟Facade类交互就好了，从而更好地实现了客户端和子系统中A、B、C模块的解耦，让客户端更容易地使用系统。

三、外观模式的扩展

使用外观模式还有一个附带的好处，就是能够有选择性地暴露方法。一个模块中定义的方法可以分成两部分，一部分是给子系统外部使用的，一部分是子系统内部模块之间相互调用时使用的。有了Facade类，那么用于子系统内部模块之间相互调用的方法就不用暴露给子系统外部了。

比如，定义如下A、B、C模块。

```
public class Module {
    /**
     * 提供给子系统外部使用的方法
     */
    public void a1(){};

    /**
     * 子系统内部模块之间相互调用时使用的方法
     */
    public void a2(){};
    public void a3(){};
}
```

```
public class ModuleB {
    /**
     * 提供给子系统外部使用的方法
     */
    public void b1(){};

    /**
     * 子系统内部模块之间相互调用时使用的方法
     */
    public void b2(){};
    public void b3(){};
}
```

```
public class ModuleC {
    /**
     * 提供给子系统外部使用的方法
     */
    public void c1(){};

    /**
     * 子系统内部模块之间相互调用时使用的方法
     */
    public void c2(){};
    public void c3(){};
}
```

```
public class ModuleFacade {

    ModuleA a = new ModuleA();
    ModuleB b = new ModuleB();
    ModuleC c = new ModuleC();
    /**
     * 下面这些是A、B、C模块对子系统外部提供的方法
     */
    public void a1(){
        a.a1();
    }
    public void b1(){
        b.b1();
    }
    public void c1(){
        c.c1();
    }
}
```

这样定义一个ModuleFacade类可以有效地屏蔽内部的细节，免得客户端去调用Module类时，发现一些不需要它知道的方法。比如a2()和a3()方法就不需要让客户端知道，否则既暴露了内部的细节，又让客户端迷惑。

一个系统可以有几个外观类

在外观模式中，通常只需要一个外观类，并且此外观类只有一个实例，换言之它是一个单例类。当然这并不意味着在整个系统里只有一个外观类，而仅仅是说对每一个子系统只有一个外观类。或者说，如果一个系统有好几个子系统的话，每一个子系统都有一个外观类，整个系统可以有数个外观类。

为子系统增加新行为

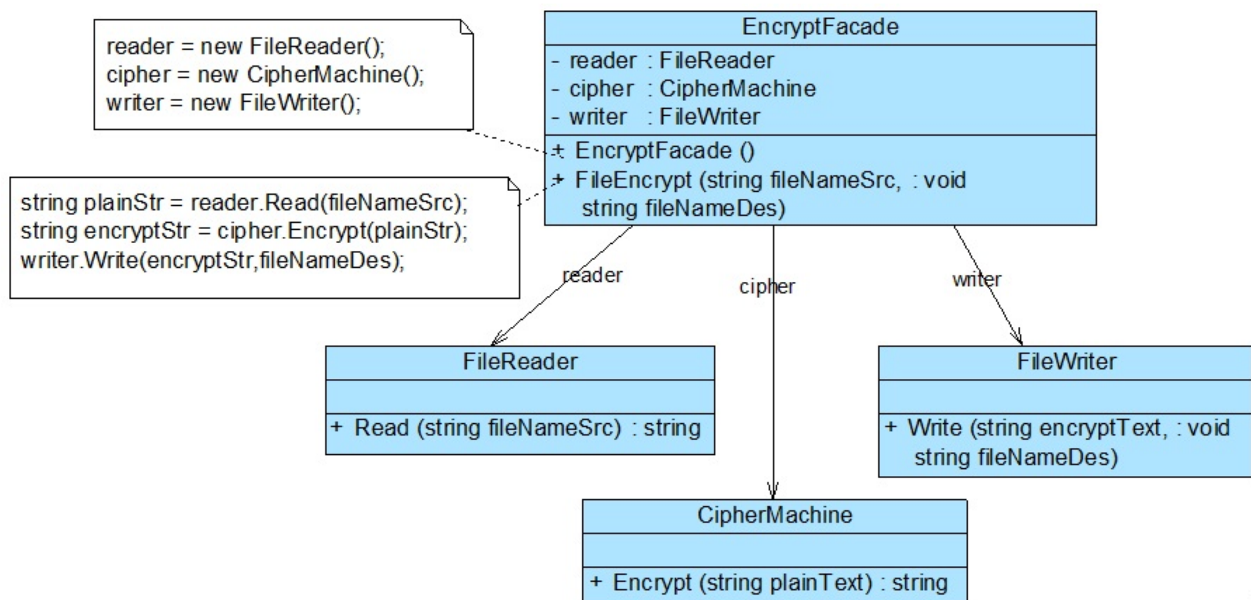
初学者往往以为通过继承一个外观类便可在子系统中加入新的行为，这是错误的。外观模式的用意是为子系统提供一个集中化和简化的沟通管道，而不能向子系统加入新的行为。比如医院中的接待员并不是医护人员，接待员并不能为病人提供医疗服务。

四、外观模式的实例

1.实例说明

某软件公司欲开发一个可应用于多个软件的文件加密模块，该模块可以对文件中的数据进行加密并将加密之后的数据存储在一个新文件中，具体的流程包括三个部分，分别是读取源文件、加密、保存加密之后的文件，其中，读取文件和保存文件使用流来实现，加密操作通过求模运算实现。这三个操作相对独立，为了实现代码的独立重用，让设计更符合单一职责原则，这三个操作的业务代码封装在三个不同的类中。

2.实例类图



EncryptFacade充当外观类，FileReader、CipherMachine和FileWriter充当子系统类。

3.实例代码

FileReader：文件读取类，充当子系统类。

```
class FileReader
    {
        public string Read(string fileNameSrc)
        {
       Console.Write("读取文件，获取明文：");
            FileStream fs = null;
            StringBuilder sb = new StringBuilder();
       try
            {
                fs = new FileStream(fileNameSrc, FileMode.Open);

                int data;
               while((data = fs.ReadByte())!= -1)
                {
            sb = sb.Append((char)data);
               }
               fs.Close();
               Console.WriteLine(sb.ToString());
       }
       catch(FileNotFoundException e)
            {
           Console.WriteLine("文件不存在！");
       }
       catch(IOException e)
            {
           Console.WriteLine("文件操作错误！");
       }
       return sb.ToString();
        }
    }
```

CipherMachine：数据加密类，充当子系统类。

```
class CipherMachine
    {
       public string Encrypt(string plainText)
       {
       Console.Write("数据加密，将明文转换为密文：");
       string es = "";
            char[] chars = plainText.ToCharArray();
       foreach(char ch in chars)
            {
                string c = (ch % 7).ToString();
           es += c;
       }
            Console.WriteLine(es);
       return es;
    }
    }
```

FileWriter：文件保存类，充当子系统类。

```
class FileWriter
    {
        public void Write(string encryptStr,string fileNameDes)

        {
       Console.WriteLine("保存密文，写入文件。");
            FileStream fs = null;
       try
            {
               fs = new FileStream(fileNameDes, FileMode.Create)
;
                byte[] str = Encoding.Default.GetBytes(encryptSt
r);
                fs.Write(str,0,str.Length);
                fs.Flush();
               fs.Close();
       }
       catch(FileNotFoundException e)
            {
        Console.WriteLine("文件不存在！");
       }
       catch(IOException e)
            {
                Console.WriteLine(e.Message);
           Console.WriteLine("文件操作错误！");
       }
        }
    }
```

EncryptFacade：加密外观类，充当外观类。

```
class EncryptFacade
    {
        //维持对其他对象的引用
         private FileReader reader;
        private CipherMachine cipher;
        private FileWriter writer;

        public EncryptFacade()
        {
            reader = new FileReader();
            cipher = new CipherMachine();
            writer = new FileWriter();
        }

        //调用其他对象的业务方法
         public void FileEncrypt(string fileNameSrc, string file
NameDes)
        {
            string plainStr = reader.Read(fileNameSrc);
            string encryptStr = cipher.Encrypt(plainStr);
            writer.Write(encryptStr, fileNameDes);
        }
    }
```

Program：客户端测试类

```
class Program
    {
        static void Main(string[] args)
        {
            EncryptFacade ef = new EncryptFacade();
            ef.FileEncrypt("src.txt", "des.txt");
            Console.Read();
        }
    }
```

五、外观模式的优点

- 松散耦合

外观模式松散了客户端与子系统的耦合关系，让子系统内部的模块能更容易扩展和维护。

- 简单易用

外观模式让子系统更加易用，客户端不再需要了解子系统内部的实现，也不需要跟众多子系统内部的模块进行交互，只需要跟外观类交互就可以了。

- 更好的划分访问层次

通过合理使用Facade，可以帮助我们更好地划分访问的层次。有些方法是对系统外的，有些方法是系统内部使用的。把需要暴露给外部的功能集中到外观中，这样既方便客户端使用，也很好地隐藏了内部的细节。

一、装饰者模式的概念

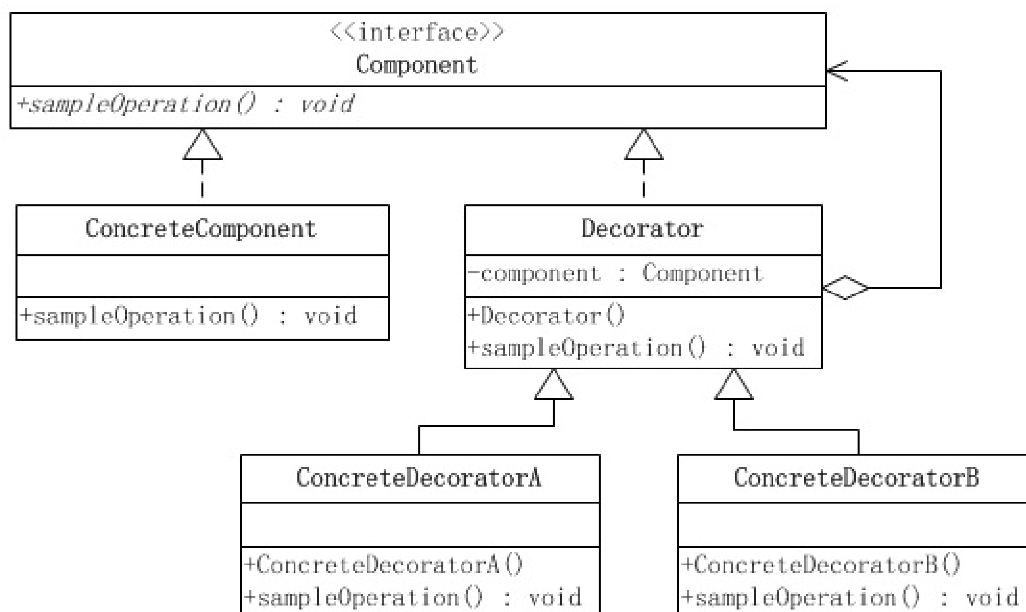
装饰者模式又名包装(Wrapper)模式。装饰者模式以对客户端透明的方式扩展对象的功能，是继承关系的一个替代方案。

装饰者模式动态地将责任附加到对象身上。若要扩展功能，装饰者提供了比继承更有弹性的替代方案。

二、装饰者模式的结构

装饰者模式以对客户透明的方式动态地给一个对象附加上更多的责任。换言之，客户端并不会觉得对象在装饰前和装饰后有什么不同。装饰者模式可以在不使用创造更多子类的情况下，将对象的功能加以扩展。

装饰者模式的类图如下：



在装饰模式中的角色有：

- 抽象构件(Component)角色：给出一个抽象接口，以规范准备接收附加责任的对象。

- 具体构件(ConcreteComponent)角色：定义一个将要接收附加责任的类。

- 装饰(Decorator)角色：持有一个构件(Component)对象的实例，并定义一个与抽象构件接口一致的接口。

- 具体装饰(ConcreteDecorator)角色：负责给构件对象“贴上”附加的责任。

抽象构件角色

```
public interface Component {

    public void sampleOperation();

}
```

具体构件角色

```
public class ConcreteComponent implements Component {

    @Override
    public void sampleOperation() {
        // 写相关的业务代码
    }
}
```

装饰角色

```
public class Decorator implements Component{
    private Component component;

    public Decorator(Component component){
        this.component = component;
    }

    @Override
    public void sampleOperation() {
        // 委派给构件
        component.sampleOperation();
    }
}
```

具体装饰角色

```
public class ConcreteDecoratorA extends Decorator {

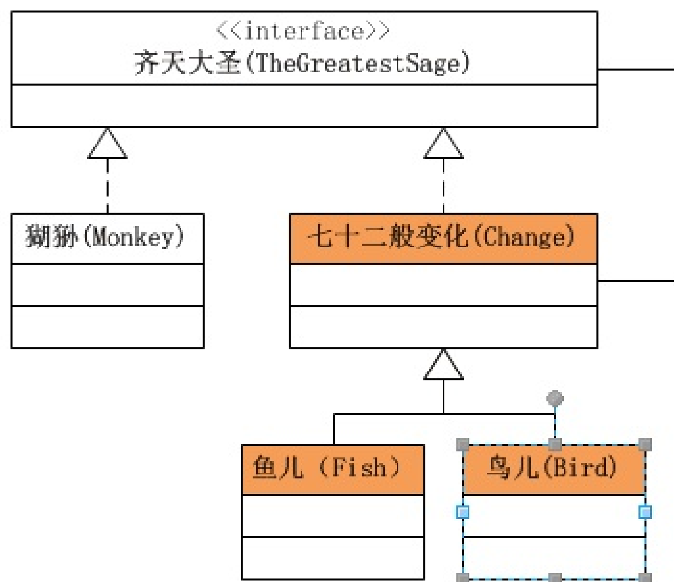
    public ConcreteDecoratorA(Component component) {
        super(component);
    }

    @Override
    public void sampleOperation() {
      // 写相关的业务代码
     super.sampleOperation();
     // 写相关的业务代码
    }
}
```

三、装饰者模式实例演示

齐天大圣的例子
孙悟空有七十二般变化，他的每一种变化都给他带来一种附加的本领。他变成鱼儿时，就可以到水里游泳；他变成鸟儿时，就可以在天上飞行。

本例中，Component的角色便由鼎鼎大名的齐天大圣扮演；ConcreteComponent的角色属于大圣的本尊，就是猢狲本人；Decorator的角色由大圣的七十二变扮演。而ConcreteDecorator的角色便是鱼儿、鸟儿等七十二般变化。



抽象构件角色“齐天大圣”接口定义了一个**move()**方法，这是所有的具体构件类和装饰类必须实现的。

```
//大圣的尊号
public interface TheGreatestSage {

    public void move();
}
```

具体构件角色“大圣本尊”猢狲类

```
public class Monkey implements TheGreatestSage {

    @Override
    public void move() {
        //代码
        System.out.println("Monkey Move");
    }
}
```

抽象装饰角色“七十二变”

```
public class Change implements TheGreatestSage {
    private TheGreatestSage sage;

    public Change(TheGreatestSage sage){
        this.sage = sage;
    }
    @Override
    public void move() {
        // 代码
        sage.move();
    }
}
```

具体装饰角色“鱼儿”

```
public class Fish extends Change {

    public Fish(TheGreatestSage sage) {
        super(sage);
    }

    @Override
    public void move() {
        // 代码
        System.out.println("Fish Move");
    }
}
```

具体装饰角色“鸟儿”

```
public class Bird extends Change {

    public Bird(TheGreatestSage sage) {
        super(sage);
    }

    @Override
    public void move() {
        // 代码
        System.out.println("Bird Move");
    }
}
```

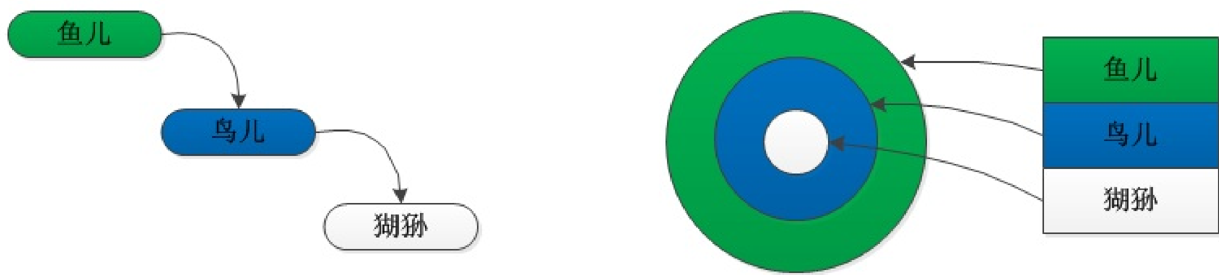
客户端调用

```
public class Client {

    public static void main(String[] args) {
        TheGreatestSage sage = new Monkey();
        // 第一种写法  单层装饰
        TheGreatestSage bird = new Bird(sage);
        TheGreatestSage fish = new Fish(bird);
        // 第二种写法 双层装饰
        //TheGreatestSage fish = new Fish(new Bird(sage));
        fish.move();
    }
}
```

“大圣本尊”是ConcreteComponent类，而“鸟儿”、“鱼儿”是装饰类。要装饰的是“大圣本尊”，也即“猢狲”实例。

上面的例子中，第二种些方法：系统把大圣从一只猢狲装饰成了一只鸟儿（把鸟儿的功能加到了猢狲身上），然后又把鸟儿装饰成了一条鱼儿（把鱼儿的功能加到了猢狲+鸟儿身上，得到了猢狲+鸟儿+鱼儿）。



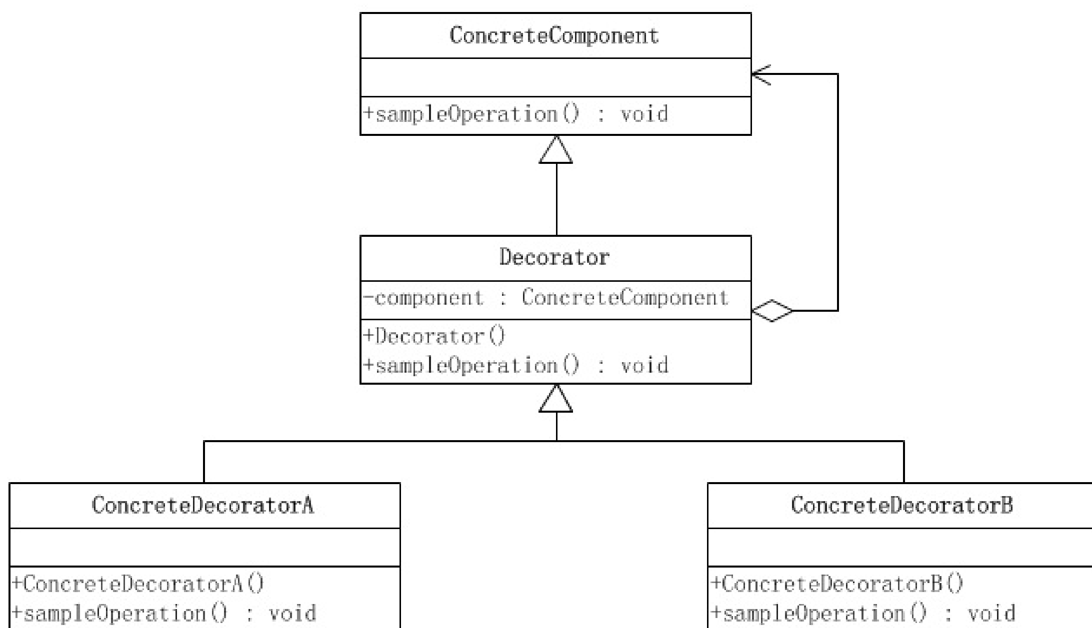
如上图所示，大圣的变化首先将鸟儿的功能附加到了猢狲身上，然后又将鱼儿的功能附加到猢狲+鸟儿身上。

四、装饰者模式的一些变化

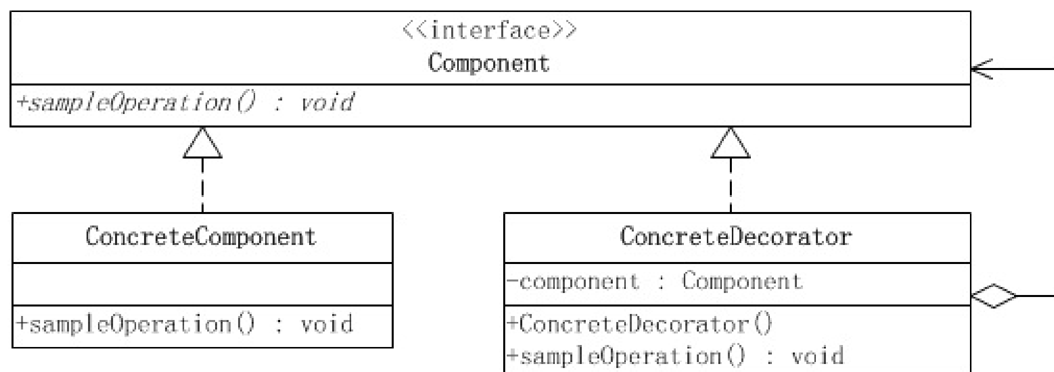
1.装饰者模式的简化

大多数情况下，装饰者模式的实现都要比上面给出的示意性例子要简单。

如果只有一个ConcreteComponent类，那么可以考虑去掉抽象的Component类（接口），把Decorator作为一个ConcreteComponent子类。如下图所示：



如果只有一个ConcreteDecorator类，那么就没有必要建立一个单独的Decorator类，而可以把Decorator和ConcreteDecorator的责任合并成一个类。甚至在只有两个ConcreteDecorator类的情况下，都可以这样做。如下图所示：



2.透明性的要求

装饰者模式对客户端的透明性要求程序不要声明一个ConcreteComponent类型的变量，而应当声明一个Component类型的变量。

用孙悟空的例子来说，必须永远把孙悟空的所有变化都当成孙悟空来对待，而如果把老孙变成的鱼儿当成鱼儿，而不是老孙，那就被老孙骗了，而这时不应当发生的。下面的做法是对的：

```
TheGreatestSage sage = new Monkey();
TheGreatestSage bird = new Bird(sage);
```

而下面的做法是不对的：

```
Monkey sage = new Monkey();
Bird bird = new Bird(sage);
```

3.半透明的装饰者模式

然而，纯粹的装饰者模式很难找到。装饰者模式的用意是在不改变接口的前提下，增强所考虑的类的性能。在增强性能的时候，往往需要建立新的公开的方法。即便是在孙大圣的系统里，也需要新的方法。比如齐天大圣类并没有飞行的能力，而鸟儿有。这就意味着鸟儿应当有一个新的fly()方法。再比如，齐天大圣类并没有游泳的能力，而鱼儿有，这就意味着在鱼儿类里应当有一个新的swim()方法。

这就导致了大多数的装饰者模式的实现都是“半透明”的，而不是完全透明的。换言之，允许装饰者模式改变接口，增加新的方法。这意味着客户端可以声明ConcreteDecorator类型的变量，从而可以调用ConcreteDecorator类中才有的方法：

```
TheGreatestSage sage = new Monkey();
Bird bird = new Bird(sage);
bird.fly();
```

半透明的装饰者模式是介于装饰者模式和适配器模式之间的。适配器模式的用意是改变所考虑的类的接口，也可以通过改写一个或几个方法，或增加新的方法来增强或改变所考虑的类的功能。大多数的装饰者模式实际上是半透明的装饰者模式，这样的装饰者模式也称做半装饰、半适配器模式。

五、装饰者模式的优缺点

装饰模式的优点

（1）装饰模式与继承关系的目的都是要扩展对象的功能，但是装饰模式可以提供比继承更多的灵活性。装饰模式允许系统动态决定“贴上”一个需要的“装饰”，或者除掉一个不需要的“装饰”。继承关系则不同，继承关系是静态的，它在系统运行前就决定了。

（2）通过使用不同的具体装饰类以及这些装饰类的排列组合，设计师可以创造出很多不同行为的组合。

装饰模式的缺点

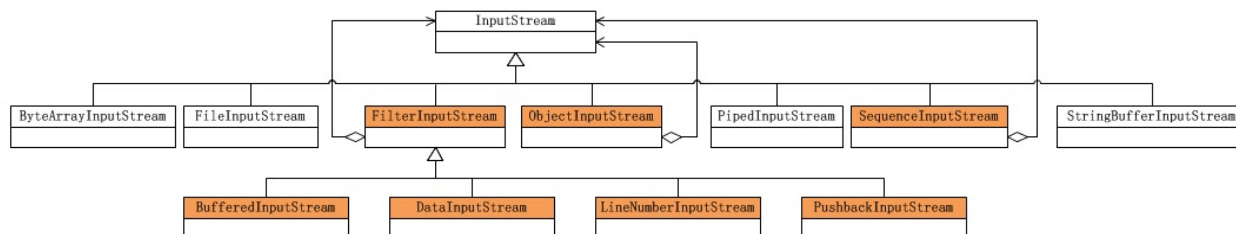
由于使用装饰模式，可以比使用继承关系需要较少数目的类。使用较少的类，当然使设计比较易于进行。但是，在另一方面，使用装饰模式会产生比使用继承关系更多的对象。更多的对象会使得查错变得困难，特别是这些对象看上去都很相像。

六、装饰者模式在Java IO流中的应用

装饰者模式在Java语言中的最著名的应用莫过于Java I/O标准库的设计了。

由于Java I/O库需要很多性能的各种组合，如果这些性能都是用继承的方法实现的，那么每一种组合都需要一个类，这样就会造成大量性能重复的类出现。而如果采用装饰者模式，那么类的数目就会大大减少，性能的重复也可以减至最少。因此装饰者模式是Java I/O库的基本模式。

Java I/O库的对象结构图如下，由于Java I/O的对象众多，因此只画出InputStream的部分。



根据上图可以看出：

- 抽象构件(Component)角色：由InputStream扮演。这是一个抽象类，为各种子类型提供统一的接口。

- 具体构件(ConcreteComponent)角色：由ByteArrayInputStream、FileInputStream、PipedInputStream、StringBufferInputStream等类扮演。它们实现了抽象构件角色所规定的接口。

- 抽象装饰(Decorator)角色：由FilterInputStream扮演。它实现了InputStream所规定的接口。

- 具体装饰(ConcreteDecorator)角色：由几个类扮演，分别是BufferedInputStream、DataInputStream以及两个不常用到的类LineNumberInputStream、PushbackInputStream。

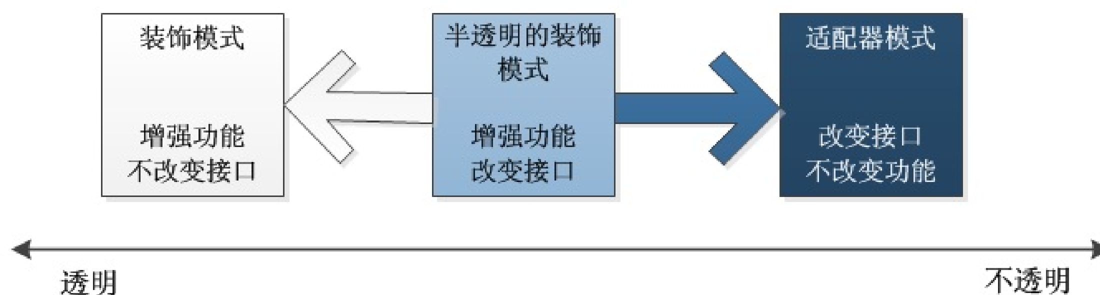
半透明的装饰者模式

装饰者模式和适配器模式都是“包装模式(Wrapper Pattern)”，它们都是通过封装其他对象达到设计的目的的，但是它们的形态有很大区别。

理想的装饰者模式在对被装饰对象进行功能增强的同时，要求具体构件角色、装饰角色的接口与抽象构件角色的接口完全一致。而适配器模式则不然，一般而言，适配器模式并不要求对源对象的功能进行增强，但是会改变源对象的接口，以便和目标接口相符合。

装饰者模式有透明和半透明两种，这两种的区别就在于装饰角色的接口与抽象构件角色的接口是否完全一致。透明的装饰者模式也就是理想的装饰者模式，要求具体构件角色、装饰角色的接口与抽象构件角色的接口完全一致。相反，如果装饰角色

的接口与抽象构件角色接口不一致，也就是说装饰角色的接口比抽象构件角色的接口宽的话，装饰角色实际上已经成了一个适配器角色，这种装饰者模式也是可以接受的，称为“半透明”的装饰模式，如下图所示。



在适配器模式里面，适配器类的接口通常会与目标类的接口重叠，但往往并不完全相同。换言之，适配器类的接口会比被装饰的目标类接口宽。

显然，半透明的装饰者模式实际上就是处于适配器模式与装饰者模式之间的灰色地带。如果将装饰者模式与适配器模式合并成为一个“包装模式”的话，那么半透明的装饰者模式倒可以成为这种合并后的“包装模式”的代表。

InputStream类型中的装饰者模式

InputStream类型中的装饰者模式是半透明的。为了说明这一点，不妨看一看作装饰者模式的抽象构件角色的InputStream的源代码。这个抽象类声明了九个方法，并给出了其中八个的实现，另外一个是抽象方法，需要由子类实现。

```
public abstract class InputStream implements Closeable {

    public abstract int read() throws IOException;


    public int read(byte b[]) throws IOException {}

    public int read(byte b[], int off, int len) throws IOExcepti
on {}

    public long skip(long n) throws IOException {}

    public int available() throws IOException {}

    public void close() throws IOException {}

    public synchronized void mark(int readlimit) {}

    public synchronized void reset() throws IOException {}

    public boolean markSupported() {}
}
```

下面是作为装饰模式的抽象装饰角色FilterInputStream类的源代码。可以看出，FilterInputStream的接口与InputStream的接口是完全一致的。也就是说，直到这一步，还是与装饰模式相符合的。

```
public class FilterInputStream extends InputStream {
    protected FilterInputStream(InputStream in) {}

    public int read() throws IOException {}

    public int read(byte b[]) throws IOException {}

    public int read(byte b[], int off, int len) throws IOExcepti
on {}

    public long skip(long n) throws IOException {}

    public int available() throws IOException {}

    public void close() throws IOException {}

    public synchronized void mark(int readlimit) {}

    public synchronized void reset() throws IOException {}

    public boolean markSupported() {}
}
```

下面是具体装饰角色PushbackInputStream的源代码。

```
public class PushbackInputStream extends FilterInputStream {
    private void ensureOpen() throws IOException {}

    public PushbackInputStream(InputStream in, int size) {}

    public PushbackInputStream(InputStream in) {}

    public int read() throws IOException {}

    public int read(byte[] b, int off, int len) throws IOExcepti
on {}

    public void unread(int b) throws IOException {}

    public void unread(byte[] b, int off, int len) throws IOExce
ption {}

    public void unread(byte[] b) throws IOException {}

    public int available() throws IOException {}

    public long skip(long n) throws IOException {}

    public boolean markSupported() {}

    public synchronized void mark(int readlimit) {}

    public synchronized void reset() throws IOException {}

    public synchronized void close() throws IOException {}
}
```

查看源码，你会发现，这个装饰类提供了额外的方法unread()，这就意味着PushbackInputStream是一个半透明的装饰类。换言 之，它破坏了理想的装饰者模式的要求。如果客户端持有一个类型为InputStream对象的引用in的话，那么如果in的真实类型是 PushbackInputStream的话，只要客户端不需要使用unread()方法，那么客户端一般没有问题。但是如果客户端必须使用这个方法，就 必须进行向下类

型转换。将in的类型转换成为PushbackInputStream之后才可能调用这个方法。但是，这个类型转换意味着客户端必须知道它 拿到的引用是指向一个类型为PushbackInputStream的对象。这就破坏了使用装饰者模式的原始用意。

现实世界与理论总归是有一段差距的。纯粹的装饰者模式在真实的系统中很难找到。一般所遇到的，都是这种半透明的装饰者模式。

下面是使用I/O流读取文件内容的简单操作示例。

```
public class IOTest {

    public static void main(String[] args) throws IOException {
        // 流式读取文件
        DataInputStream dis = null;
        try{
            dis = new DataInputStream(
                    new BufferedInputStream(
                            new FileInputStream("test.txt")
                    )
            );
            //读取文件内容
            byte[] bs = new byte[dis.available()];
            dis.read(bs);
            String content = new String(bs);
            System.out.println(content);
        }finally{
            dis.close();
        }
    }
}
```

观察上面的代码，会发现最里层是一个FileInputStream对象，然后把它传递给一个BufferedInputStream对象，经过BufferedInputStream处理，再把处理后的对象传递给了DataInputStream对象进行处理，这个过程其实就是装饰器的组装过程，FileInputStream对象相当于原始的被装饰的对象，而BufferedInputStream对象和DataInputStream对象则相当于装饰器。

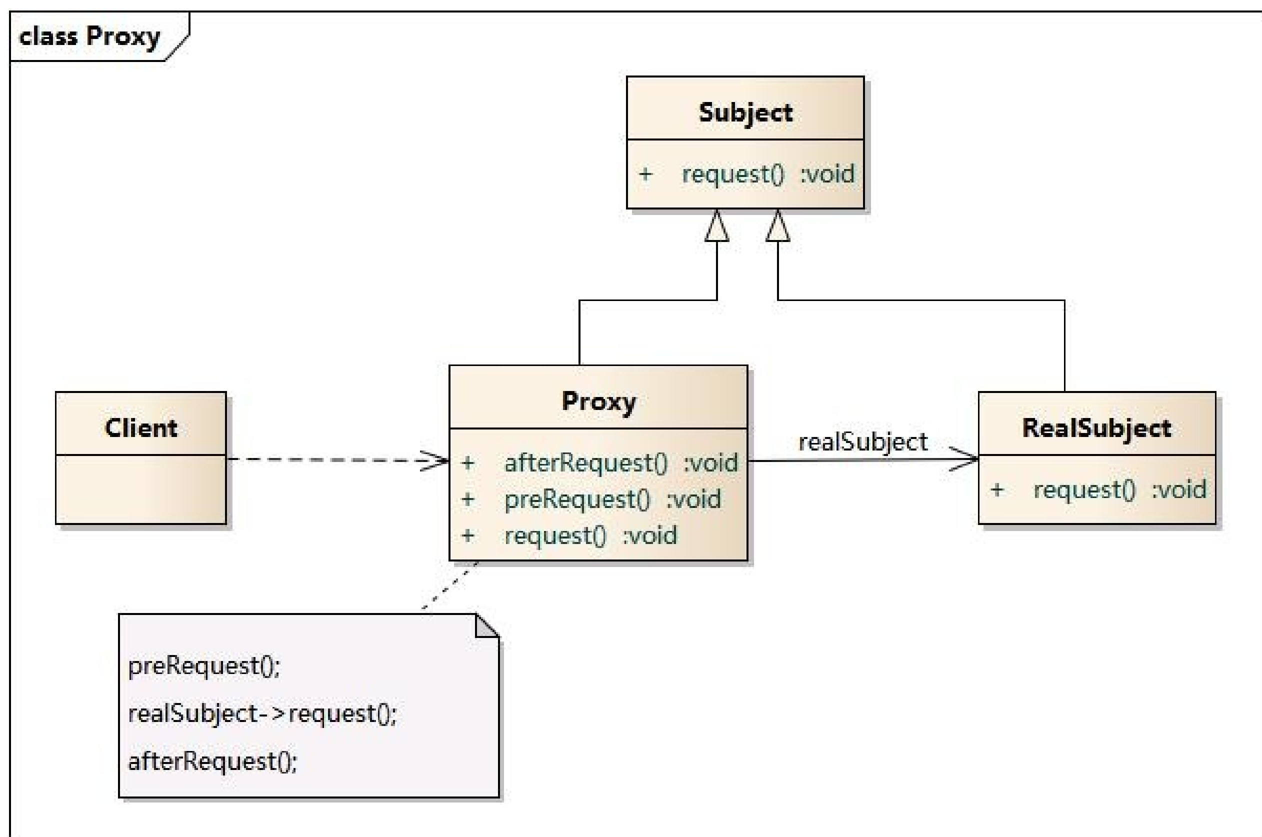
一、模式定义

代理模式(Proxy Pattern) ：给某一个对象提供一个代 理，并由代理对象控制对原对象的引用。代理模式的英 文叫做Proxy或Surrogate，它是一种对象结构型模式。

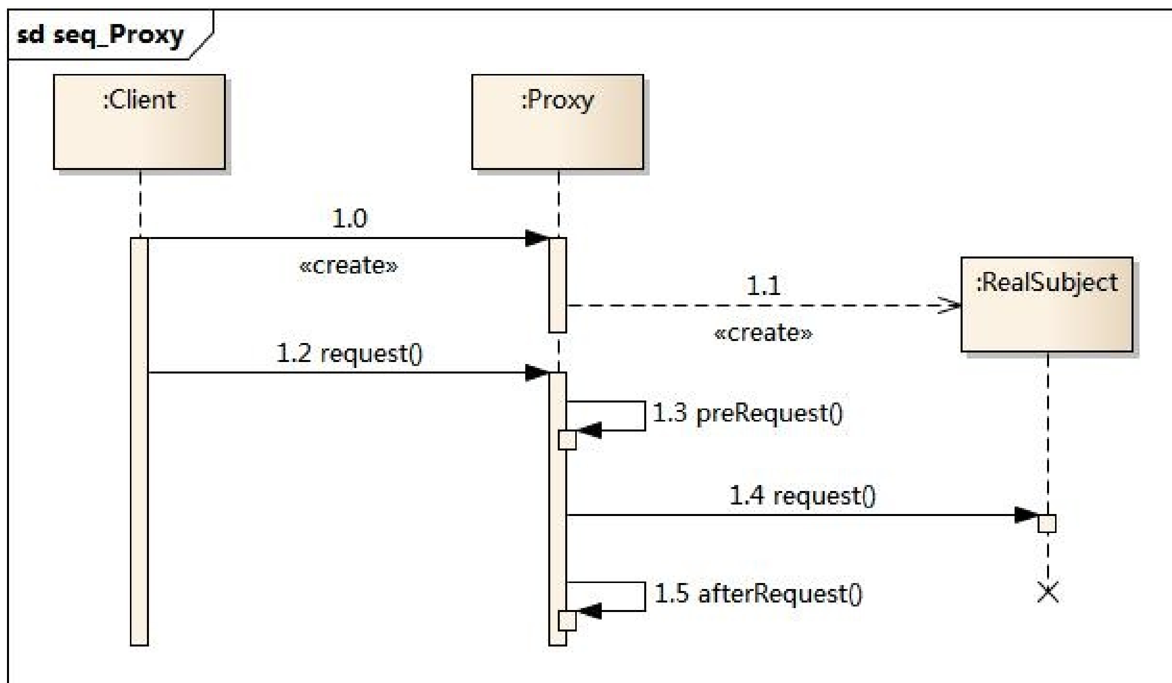
二、模式结构

代理模式包含如下角色：

- Subject: 抽象主题角色
- Proxy: 代理主题角色
- RealSubject: 真实主题角色



三、时序图



四、模式的简单实现

抽象对象角色

```
public abstract class AbstractObject {
    //操作
    public abstract void operation();
}
```

目标对象角色

```
public class RealObject extends AbstractObject {
    @Override
    public void operation() {
        //一些操作
        System.out.println("一些操作");
    }
}
```

代理对象角色

```
public class ProxyObject extends AbstractObject{
    RealObject realObject = new RealObject();
    @Override
    public void operation() {
        //调用目标对象之前可以做相关操作
        System.out.println("before");
        realObject.operation();
        //调用目标对象之后可以做相关操作
        System.out.println("after");
    }
}
```

客户端

```
public class Client {
    public static void main(String[] args) {
        AbstractObject obj = new ProxyObject();
        obj.operation();
    }
}
```

五、代理模式在Binder中的使用

直观来说，Binder是Android中的一个类，它继承了IBinder接口。从IPC角度来说，Binder是Android中的一种跨进程通信方式，Binder还可以理解为一种虚拟的物理设备，它的设备驱动是/dev/binder，该通信方式在linux中没有；从Android Framework角度来说，Binder是ServiceManager连接各种Manager（ActivityManager、WindowManager，etc）和相应ManagerService的桥梁；从Android应用层来说，Binder是客户端和服务端进行通信的媒介，当你bindService的时候，服务端会返回一个包含了服务端业务调用的Binder对象，通过这个Binder对象，客户端就可以获取服务端提供的服务或者数据，这里的服务包括普通服务和基于AIDL的服务。

Binder一个很重要的作用是：将客户端的请求参数通过Parcel包装后传到远程服务端，远程服务端解析数据并执行对应的操作，同时客户端线程挂起，当服务端方法执行完毕后，再将返回结果写入到另外一个Parcel中并将其通过Binder传回到客户

端，客户端接收到返回数据的Parcel后，Binder会解析数据包中的内容并将原始结果返回给客户端，至此，整个Binder的工作过程就完成了。由此可见，Binder更像一个数据通道，Parcel对象就在这个通道中跨进程传输，至于双方如何通信，这并不负责，只需要双方按照约定好的规范去打包和解包数据即可。

为了更好地说明Binder，这里我们先手动实现了一个Binder。为了使得逻辑更清晰，这里简化一下，我们来模拟一个银行系统，这个银行提供的功能只有一个：即查询余额，只有传递一个int的id过来，银行就会将你的余额设置为id*10，满足下大家的发财梦。

1. 先定义一个Binder接口

```
package com.ryg.design.manualbinder;

import android.os.IBinder;
import android.os.IInterface;
import android.os.RemoteException;

public interface IBank extends IInterface {

   static final String DESCRIPTOR = "com.ryg.design.manualbinder
.IBank";

   static final int TRANSACTION_queryMoney = (IBinder.FIRST_CALL
_TRANSACTION + 0);

   public long queryMoney(int uid) throws RemoteException;

}
```

2.创建一个Binder并实现这个上述接口

```
package com.ryg.design.manualbinder;

import android.os.Binder;
import android.os.IBinder;
import android.os.Parcel;
import android.os.RemoteException;
```

```
public class BankImpl extends Binder implements IBank {

    public BankImpl() {
        this.attachInterface(this, DESCRIPTOR);
    }

    public static IBank asInterface(IBinder obj) {
        if ((obj == null)) {
            return null;
        }
        android.os.IInterface iin = obj.queryLocalInterface(DESC
RIPTOR);
        if (((iin != null) && (iin instanceof IBank))) {
            return ((IBank) iin);
        }
        return new BankImpl.Proxy(obj);
    }

    @Override
    public IBinder asBinder() {
        return this;
    }

    @Override
    public boolean onTransact(int code, Parcel data, Parcel repl
y, int flags)
            throws RemoteException {
        switch (code) {
        case INTERFACE_TRANSACTION: {
            reply.writeString(DESCRIPTOR);
            return true;
        }
        case TRANSACTION_queryMoney: {
            data.enforceInterface(DESCRIPTOR);
            int uid = data.readInt();
            long result = this.queryMoney(uid);
            reply.writeNoException();
            reply.writeLong(result);
            return true;
        }
```

```
        }
        return super.onTransact(code, data, reply, flags);
    }

    @Override
    public long queryMoney(int uid) throws RemoteException {
        return uid * 10l;
    }

    private static class Proxy implements IBank {
        private IBinder mRemote;

        Proxy(IBinder remote) {
            mRemote = remote;
        }

        @Override
        public IBinder asBinder() {
            return mRemote;
        }

        public java.lang.String getInterfaceDescriptor() {
            return DESCRIPTOR;
        }

        @Override
        public long queryMoney(int uid) throws RemoteException {
            Parcel data = Parcel.obtain();
            Parcel reply = Parcel.obtain();
            long result;
            try {
                data.writeInterfaceToken(DESCRIPTOR);
                data.writeInt(uid);
                mRemote.transact(TRANSACTION_queryMoney, data, r
eply, 0);
                reply.readException();
                result = reply.readLong();
            } finally {
                reply.recycle();
                data.recycle();
```

```
            }
            return result;
        }

    }

}
```

ok，到此为止，我们的Binder就完成了，这里只要创建服务端和客户端，二者就能通过我们的Binder来通信了。这里就不做这个示例了，我们的目的是分析代理模式在Binder中的使用。

我们看上述Binder的实现中，有一个叫做“Proxy”的类，它的构造方法如下：

```
Proxy(IBinder remote) {
    mRemote = remote;
}
```

Proxy类接收一个IBinder参数，这个参数实际上就是服务端Service中的onBind方法返回的Binder对象在客户端重新打包后的结果，因为客户端无法直接通过这个打包的Binder和服务端通信，因此客户端必须借助Proxy类来和服务端通信，这里Proxy的作用就是代理的作用，客户端所有的请求全部通过Proxy来代理，具体工作流程为：Proxy接收到客户端的请求后，会将客户端的请求参数打包到Parcel对象中，然后将Parcel对象通过它内部持有的Ibinder对象传送到服务端，服务端接收数据、执行方法后返回结果给客户端的Proxy，Proxy解析数据后返回给客户端的真正调用者。很显然，上述所分析的就是典型的代理模式。至于Binder如何传输数据，这涉及到很底层的知识，这个很难搞懂，但是数据传输的核心思想是共享内存。

六、优缺点

优点

- 给对象增加了本地化的扩展性，增加了存取操作控制

缺点

- 会产生多余的代理类

一、前言

行为型模式

行为型模式(Behavioral Pattern)是对在不同的对象之间划分责任和算法的抽象化。

行为型模式不仅仅关注类和对象的结构，而且重点关注它们之间的相互作用。

通过行为型模式，可以更加清晰地划分类与对象的职责，并研究系统在运行时实例对象 之间的交互。在系统运行时，对象并不是孤立的，它们可以通过相互通信与协作完成某些复杂功能，一个对象在运行时也将影响到其他对象的运行。

行为型模式分为类行为型模式和对象行为型模式两种：

- 类行为型模式：类的行为型模式使用继承关系在几个类之间分配行为，类行为型模式主要通过多态等方式来分配父类与子类的职责。
- 对象行为型模式：对象的行为型模式则使用对象的聚合关联关系来分配行为，对象行为型模式主要是通过对象关联等方式来分配两个或多个类的职责。根据“合成复用原则”，系统中要尽量使用关联关系来取代继承关系，因此大部分行为型设计模式都属于对象行为型设计模式。

包含模式

- 职责链模式**(Chain of Responsibility)**
- 命令模式**(Command)**
- 解释器模式**(Interpreter)**
- 迭代器模式**(Iterator)**
- 中介者模式**(Mediator)**
- 备忘录模式**(Memento)**
- 观察者模式**(Observer)**
- 状态模式**(State)**
- 策略模式**(Strategy)**
- 模板方法模式**(Template Method)**
- 访问者模式**(Visitor)**

二、目录

本部分没有包含以上所有模式，仅介绍了几种常用的。

- 命令模式
- 迭代器模式
- 观察者模式
- 策略模式
- 模板方法模式

一、命令模式的概念

命令模式属于对象的行为模式。命令模式又称为行动(Action)模式或交易(Transaction)模式。

命令模式把一个请求或者操作封装到一个对象中。命令模式允许系统使用不同的请求把客户端参数化，具有请求排队或者记录请求日志，提供命令的撤销和恢复的功能。

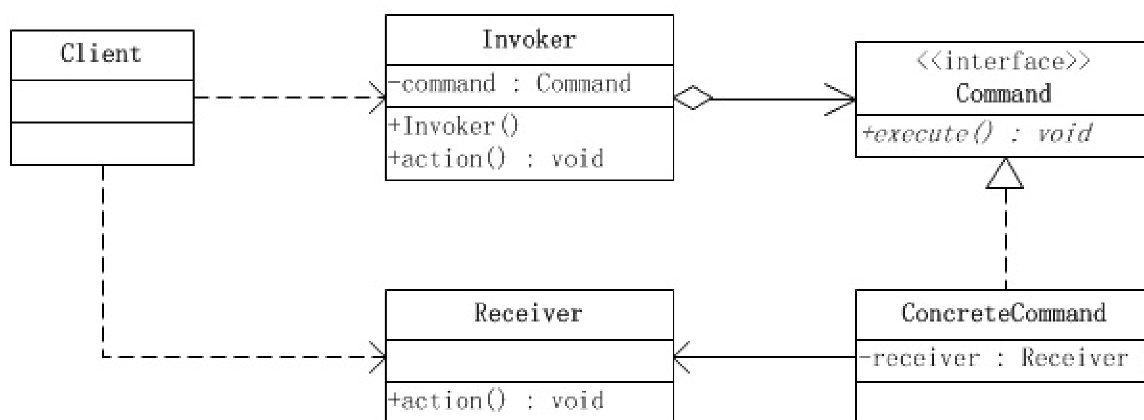
命令模式可以将请求发送者和接收者完全解耦，发送者与接收者之间没有直接引用关系，发送请求的对象只需要知道如何发送请求，而不必知道如何完成请求。

二、命令模式的结构

命令模式是对命令的封装。命令模式把发出命令的责任和执行命令的责任分割开，委派给不同的对象。

每一个命令都是一个操作：请求的一方发出请求要求执行一个操作；接收的一方收到请求，并执行操作。命令模式允许请求的一方和接收的一方独立开来，使得请求的一方不必知道接收请求的一方的接口，更不必知道请求是怎么被接收，以及操作是否被执行、何时被执行，以及是怎么被执行的。

下面以一个示意性的系统，说明命令模式的结构。



命令模式涉及到五个角色，它们分别是：

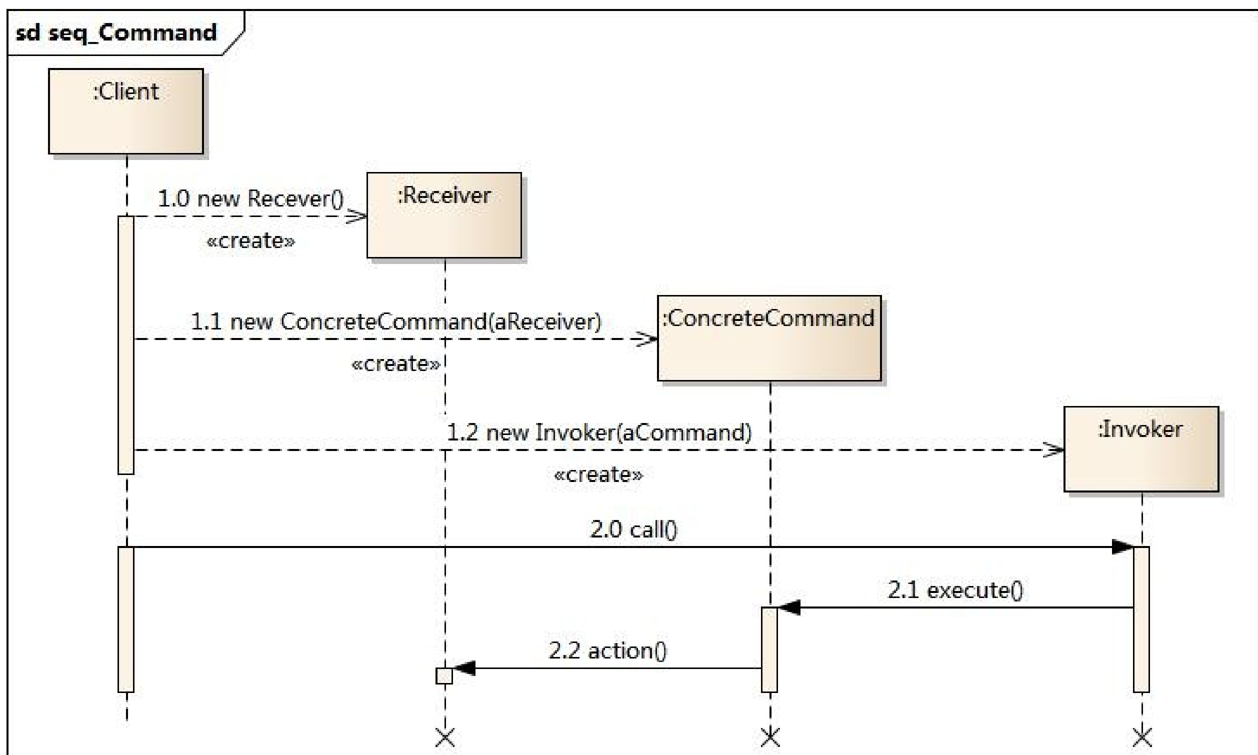
- 客户端(Client)角色：创建请求者，接收者以及命令对象，执行具体逻辑。
- 命令(Command)角色：声明了一个给所有具体命令类的抽象接口。

- 具体命令(ConcreteCommand)角色：定义一个接收者和行为之间的弱耦合；实现execute()方法，负责调用接收者的相应操作。execute()方法通常叫做执行方法。

- 请求者(Invoker)角色：负责调用命令对象执行请求，相关的方法叫做行动方法。

- 接收者(Receiver)角色：负责具体实施和执行一个请求。任何一个类都可以成为接收者，实施和执行请求的方法叫做行动方法。

时序图



接收者角色类

```
public class Receiver {
    /**
     * 真正执行命令相应的操作
     */
    public void action(){
        System.out.println("执行操作");
    }
}
```

抽象命令角色类

```
public interface Command {
    /**
     * 执行方法
     */
    void execute();
}
```

具体命令角色类

```
public class ConcreteCommand implements Command {
    //持有相应的接收者对象
    private Receiver receiver = null;
    /**
     * 构造方法
     */
    public ConcreteCommand(Receiver receiver){
        this.receiver = receiver;
    }
    @Override
    public void execute() {
        //通常会转调接收者对象的相应方法，让接收者来真正执行功能
        receiver.action();
    }

}
```

请求者角色类

```
public class Invoker {
    /**
     * 持有命令对象
     */
    private Command command = null;
    /**
     * 构造方法
     */
    public Invoker(Command command){
        this.command = command;
    }
    /**
     * 行动方法
     */
    public void action(){

        command.execute();
    }
}
```

客户端角色类

```
public class Client {

    public static void main(String[] args) {
        //创建接收者
        Receiver receiver = new Receiver();
        //创建命令对象，设定它的接收者
        Command command = new ConcreteCommand(receiver);
        //创建请求者，把命令对象设置进去
        Invoker invoker = new Invoker(command);
        //执行方法
        invoker.action();
    }

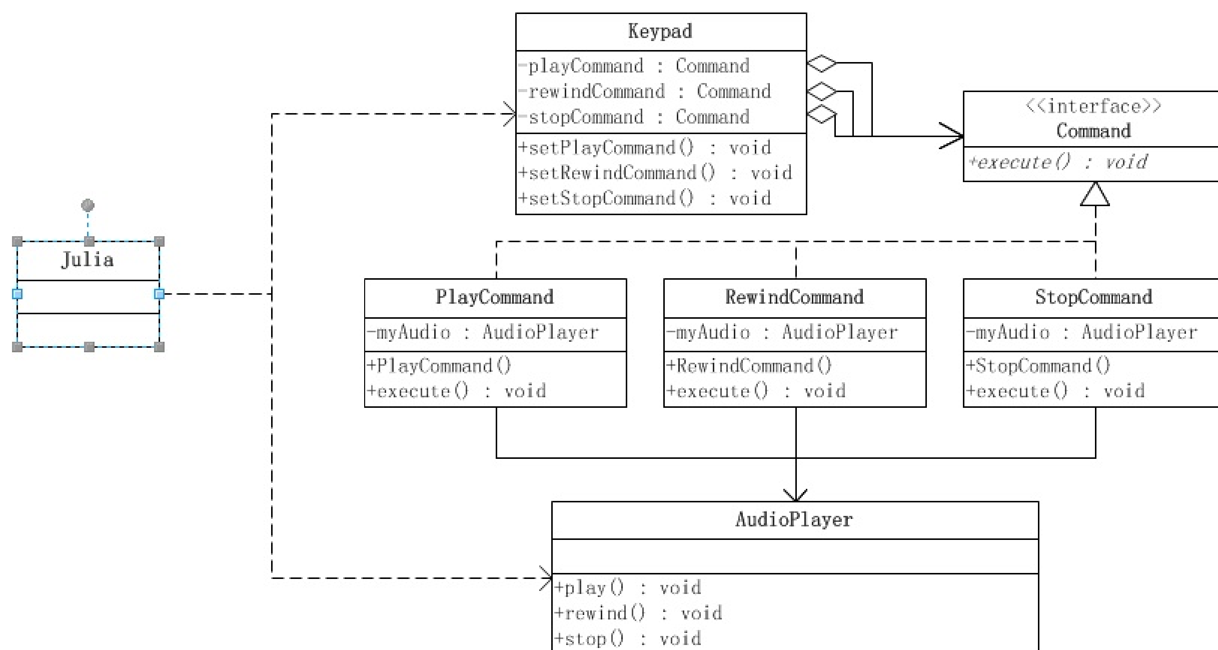
}
```

三、命令模式的具体实例

AudioPlayer系统

小女孩茱丽(Julia)有一个盒式录音机，此录音机有播音(Play)、倒带(Rewind)和停止(Stop)功能，录音机的键盘便是请求者(Invoker)角色；茱丽(Julia)是客户端角色，而录音机便是接收者角色。Command类扮演抽象命令角色，而PlayCommand、StopCommand和RewindCommand便是具体命令类。茱丽(Julia)不需要知道播音(play)、倒带(rewind)和停止(stop)功能是怎么具体执行的，这些命令执行的细节全都由键盘(Keypad)具体实施。茱丽(Julia)只需要在键盘上按下相应的键便可以了。

录音机是典型的命令模式。录音机按键把客户端与录音机的操作细节分割开来。



接收者角色，由录音机类扮演

```
public class AudioPlayer {

    public void play(){
        System.out.println("播放...");
    }

    public void rewind(){
        System.out.println("倒带...");
    }

    public void stop(){
        System.out.println("停止...");
    }
}
```

抽象命令角色类

```
public interface Command {
    /**
     * 执行方法
     */
    public void execute();
}
```

具体命令角色类

```java
public class PlayCommand implements Command {

    private AudioPlayer myAudio;

    public PlayCommand(AudioPlayer audioPlayer){
        myAudio = audioPlayer;
    }
    /**
     * 执行方法
     */
    @Override
    public void execute() {
        myAudio.play();
    }

}
```

```java
public class RewindCommand implements Command {

    private AudioPlayer myAudio;

    public RewindCommand(AudioPlayer audioPlayer){
        myAudio = audioPlayer;
    }
    @Override
    public void execute() {
        myAudio.rewind();
    }

}
```

```
public class StopCommand implements Command {
    private AudioPlayer myAudio;

    public StopCommand(AudioPlayer audioPlayer){
        myAudio = audioPlayer;
    }
    @Override
    public void execute() {
        myAudio.stop();
    }

}
```

请求者角色，由键盘类扮演

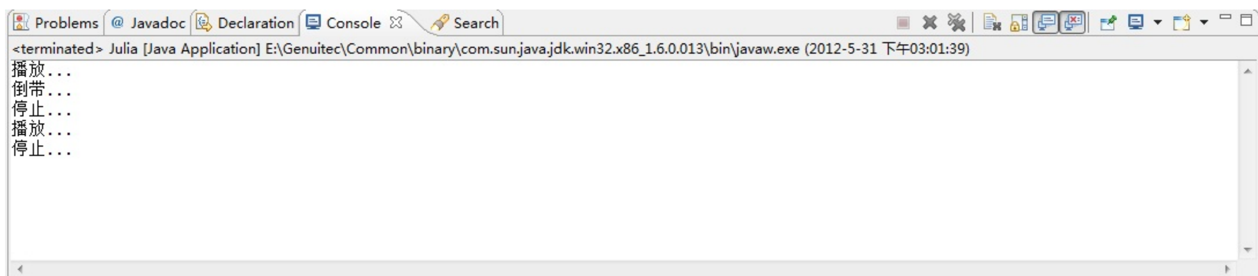
```
public class Keypad {
    private Command playCommand;
    private Command rewindCommand;
    private Command stopCommand;

    public void setPlayCommand(Command playCommand) {
        this.playCommand = playCommand;
    }
    public void setRewindCommand(Command rewindCommand) {
        this.rewindCommand = rewindCommand;
    }
    public void setStopCommand(Command stopCommand) {
        this.stopCommand = stopCommand;
    }
    /**
     * 执行播放方法
     */
    public void play(){
        playCommand.execute();
    }
    /**
     * 执行倒带方法
     */
    public void rewind(){
        rewindCommand.execute();
    }
    /**
     * 执行播放方法
     */
    public void stop(){
        stopCommand.execute();
    }
}
```

客户端角色，由茱丽小女孩扮演

```
public class Julia {
    public static void main(String[]args){
        //创建接收者对象
        AudioPlayer audioPlayer = new AudioPlayer();
        //创建命令对象
        Command playCommand = new PlayCommand(audioPlayer);
        Command rewindCommand = new RewindCommand(audioPlayer);
        Command stopCommand = new StopCommand(audioPlayer);
        //创建请求者对象
        Keypad keypad = new Keypad();
        keypad.setPlayCommand(playCommand);
        keypad.setRewindCommand(rewindCommand);
        keypad.setStopCommand(stopCommand);
        //测试
        keypad.play();
        keypad.rewind();
        keypad.stop();
        keypad.play();
        keypad.stop();
    }
}
```

运行结果：

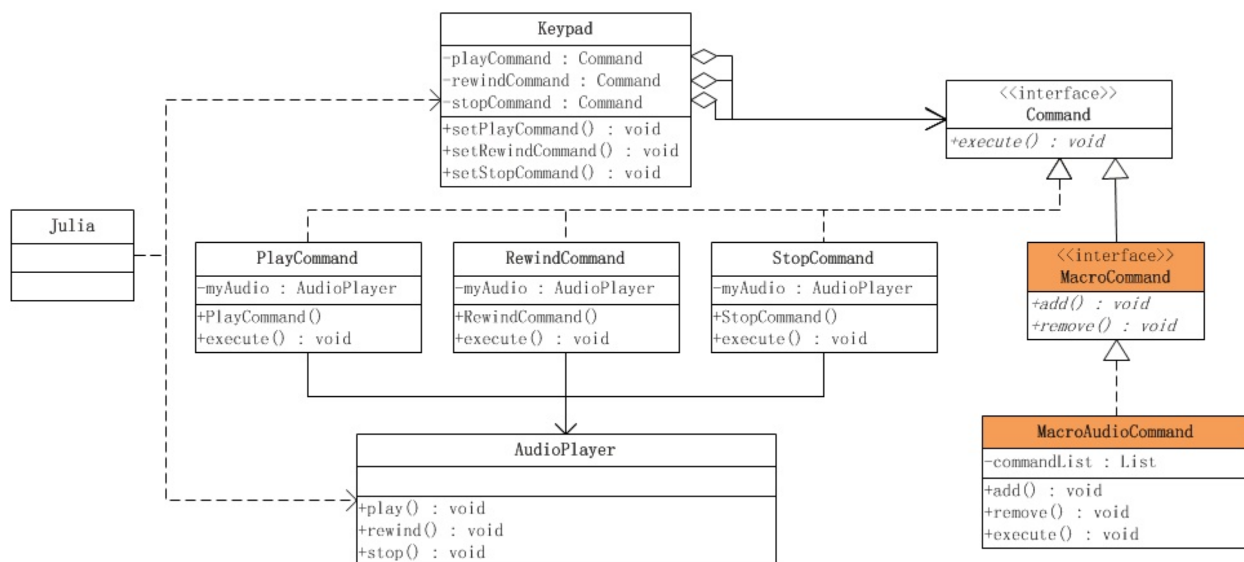


四、宏命令

宏命令
所谓宏命令简单点说就是包含多个命令的命令，是一个命令的组合。

设想茱丽的录音机有一个记录功能，可以把一个一个的命令记录下来，再在任何需要的时候重新把这些记录下来的命令一次性执行，这就是所谓的宏命令集功能。因此，茱丽的录音机系统现在有四个键，分别为播音、倒带、停止和宏命令功能。此

时系统的设计与前面的设计相比有所增强，主要体现在Julia类现在有了一个新方法，用以操作宏命令键。



系统需要一个代表宏命令的接口，以定义出具体宏命令所需要的接口。

```
public interface MacroCommand extends Command {
    /**
     * 宏命令聚集的管理方法
     * 可以添加一个成员命令
     */
    public void add(Command cmd);
    /**
     * 宏命令聚集的管理方法
     * 可以删除一个成员命令
     */
    public void remove(Command cmd);
}
```

具体的宏命令**MacroAudioCommand**类负责把个别的命令合成宏命令。

```
public class MacroAudioCommand implements MacroCommand {

    private List
<
Command
>
 commandList = new ArrayList
<
Command
>
();
    /**
     * 宏命令聚集管理方法
     */
    @Override
    public void add(Command cmd) {
        commandList.add(cmd);
    }
    /**
     * 宏命令聚集管理方法
     */
    @Override
    public void remove(Command cmd) {
        commandList.remove(cmd);
    }
    /**
     * 执行方法
     */
    @Override
    public void execute() {
        for(Command cmd : commandList){
            cmd.execute();
        }
    }

}
```

客户端类**Julia**

```
public class Julia {

    public static void main(String[]args){
        //创建接收者对象
        AudioPlayer audioPlayer = new AudioPlayer();
        //创建命令对象
        Command playCommand = new PlayCommand(audioPlayer);
        Command rewindCommand = new RewindCommand(audioPlayer);
        Command stopCommand = new StopCommand(audioPlayer);

        MacroCommand marco = new MacroAudioCommand();

        marco.add(playCommand);
        marco.add(rewindCommand);
        marco.add(stopCommand);
        marco.execute();
    }
}
```

这样执行MacroCommand 的execute()方法就会一次性执行多条命令。

五、命令模式的优缺点

优点

- 更松散的耦合

命令模式使得发起命令的对象，和具体实现命令的对象完全解耦，也就是说发起命令的对象完全不知道具体实现对象是谁，也不知道如何实现。

- 更动态的控制

命令模式把请求封装起来，可以动态地对它进行参数化、队列化和日志化等操作，从而使得系统更灵活。

- 很自然的复合命令

命令模式中的命令对象能够很容易地组合成复合命令，也就是宏命令，从而使系统操作更简单，功能更强大。

- 更好的扩展性

由于发起命令的对象和具体的实现完全解耦，因此扩展新的命令就很容易，只需要实现新的命令对象，然后在装配的时候，把具体的实现对象设置到命令对象中，然后就可以使用这个命令对象，已有的实现完全不用变化。

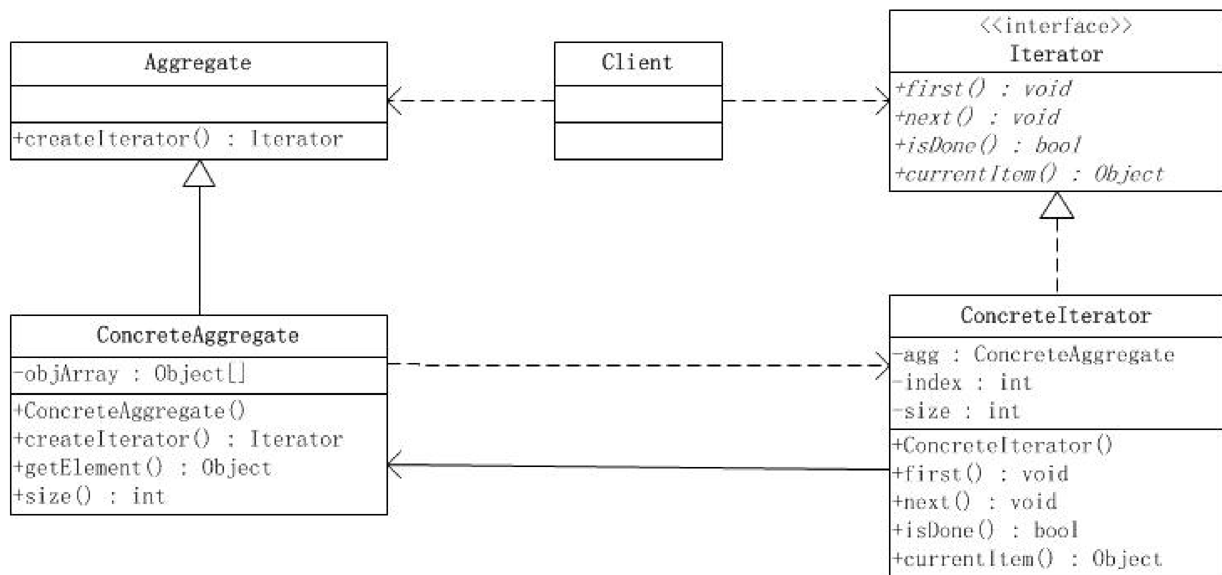
缺点

使用命令模式可能会导致某些系统有过多的具体命令类。因为针对每一个命令都需要设计一个具体命令类，因此某些系统可能需要大量具体命令类，这将影响命令模式的使用。

一、迭代器模式定义

迭代器模式提供一种方法顺序访问一个聚合对象中的各个元素，而又不暴露其内部的表示。把游走的任务放在迭代器上，而不是聚合上。这样简化了聚合的接口和实现，也让责任各得其所。

二、迭代器模式结构



迭代器模式涉及到以下几个角色：

- 抽象迭代器(Iterator)角色：此抽象角色定义出遍历元素所需的接口。

- 具体迭代器(ConcreteIterator)角色：此角色实现了Iterator接口，并保持迭代过程中的游标位置。

- 聚集(Aggregate)角色：此抽象角色给出创建迭代器(Iterator)对象的接口。

- 具体聚集(ConcreteAggregate)角色：实现了创建迭代器(Iterator)对象的接口，返回一个合适的具体迭代器实例。

- 客户端(Client)角色：持有对聚集及其迭代器对象的引用，调用迭代子对象的迭代接口，也有可能通过迭代子操作聚集元素的增加和删除。

抽象聚集角色类，这个角色规定出所有的具体聚集必须实现的接口。迭代器模式要求聚集对象必须有一个工厂方法，也就是**createIterator()**方法，以向外界提供迭代器对象的实例。

```
public abstract class Aggregate {
    /**
     * 工厂方法，创建相应迭代子对象的接口
     */
    public abstract Iterator createIterator();
}
```

具体聚集角色类，实现了抽象聚集角色类所要求的接口，也就是**createIterator()**方法。此外，还有方法**getElement()**向外界提供聚集元素，而方法**size()**向外界提供聚集的大小等。

```
public class ConcreteAggregate extends Aggregate {

    private Object[] objArray = null;
    /**
     * 构造方法，传入聚合对象的具体内容
     */
    public ConcreteAggregate(Object[] objArray){
        this.objArray = objArray;
    }

    @Override
    public Iterator createIterator() {

        return new ConcreteIterator(this);
    }
    /**
     * 取值方法：向外界提供聚集元素
     */
    public Object getElement(int index){

        if(index
<
 objArray.length){
            return objArray[index];
        }else{
            return null;
        }
    }
    /**
     * 取值方法：向外界提供聚集的大小
     */
    public int size(){
        return objArray.length;
    }
}
```

抽象迭代器角色类

```
public interface Iterator {
    /**
     * 迭代方法：移动到第一个元素
     */
    public void first();
    /**
     * 迭代方法：移动到下一个元素
     */
    public void next();
    /**
     * 迭代方法：是否为最后一个元素
     */
    public boolean isDone();
    /**
     * 迭代方法：返还当前元素
     */
    public Object currentItem();
}
```

具体迭代器角色类

```
public class ConcreteIterator implements Iterator {
    //持有被迭代的具体的聚合对象
    private ConcreteAggregate agg;
    //内部索引，记录当前迭代到的索引位置
    private int index = 0;
    //记录当前聚集对象的大小
    private int size = 0;

    public ConcreteIterator(ConcreteAggregate agg){
        this.agg = agg;
        this.size = agg.size();
        index = 0;
    }
    /**
     * 迭代方法：返还当前元素
     */
    @Override
    public Object currentItem() {
```

```
            return agg.getElement(index);
        }
        /**
         * 迭代方法：移动到第一个元素
         */
        @Override
        public void first() {

            index = 0;
        }
        /**
         * 迭代方法：是否为最后一个元素
         */
        @Override
        public boolean isDone() {
            return (index
    >
    = size);
        }
        /**
         * 迭代方法：移动到下一个元素
         */
        @Override
        public void next() {

            if(index
    <
     size)
            {
                index ++;
            }
        }

    }
```

客户端类

```
public class Client {

    public void operation(){
        Object[] objArray = {"One","Two","Three","Four","Five","
Six"};
        //创建聚合对象
        Aggregate agg = new ConcreteAggregate(objArray);
        //循环输出聚合对象中的值
        Iterator it = agg.createIterator();
        while(!it.isDone()){
            System.out.println(it.currentItem());
            it.next();
        }
    }
    public static void main(String[] args) {

        Client client = new Client();
        client.operation();
    }

}
```

三、迭代器模式的应用

如果要问Java中使用最多的一种模式，答案不是单例模式，也不是工厂模式，更不是策略模式，而是迭代器模式，先来看一段代码吧：

```
public static void print(Collection coll){
    Iterator it = coll.iterator();
    while(it.hasNext()){
        String str = (String)it.next();
        System.out.println(str);
    }
}
```

这个方法的作用是循环打印一个字符串集合，里面就用到了迭代器模式，java语言已经完整地实现了迭代器模式，例如List，Set，Map，而迭代器的作用就是把容器中的对象一个一个地遍历出来。

四、迭代器模式的优缺点

优点

①简化了遍历方式，对于对象集合的遍历，还是比较麻烦的，对于数组或者有序列表，我们尚可以通过游标来取得，但用户需要在对集合了解很清楚的前提下，自行遍历对象，但是对于hash表来说，用户遍历起来就比较麻烦了。而引入了迭代器方法后，用户用起来就简单的多了。

②可以提供多种遍历方式，比如说对有序列表，我们可以根据需要提供正序遍历，倒序遍历两种迭代器，用户用起来只需要得到我们实现好的迭代器，就可以方便的对集合进行遍历了。

③封装性良好，用户只需要得到迭代器就可以遍历，而对于遍历算法则不用去关心。

缺点

对于比较简单的遍历（像数组或者有序列表），使用迭代器方式遍历较为繁琐，大家可能都有感觉，像ArrayList，我们宁可愿意使用for循环和get方法来遍历集合。

五、迭代器的应用场景

迭代器模式是与集合共生共死的，一般来说，我们只要实现一个集合，就需要同时提供这个集合的迭代器，就像java中的Collection，List、Set、Map等，这些集合都有自己的迭代器。假如我们要实现一个这样的新的容器，当然也需要引入迭代器模式，给我们的容器实现一个迭代器。

但是，由于容器与迭代器的关系太密切了，所以大多数语言在实现容器的时候都给提供了迭代器，并且这些语言提供的容器和迭代器在绝大多数情况下就可以满足我们的需要，所以现在需要我们自己去实践迭代器模式的场景还是比较少见的，我们只需要使用语言中已有的容器和迭代器就可以了。

一、观察者模式的概念

观察者模式是对象的行为模式，又叫发布-订阅(Publish/Subscribe)模式、模型-视图(Model/View)模式、源-监听器(Source/Listener)模式或从属者(Dependents)模式。

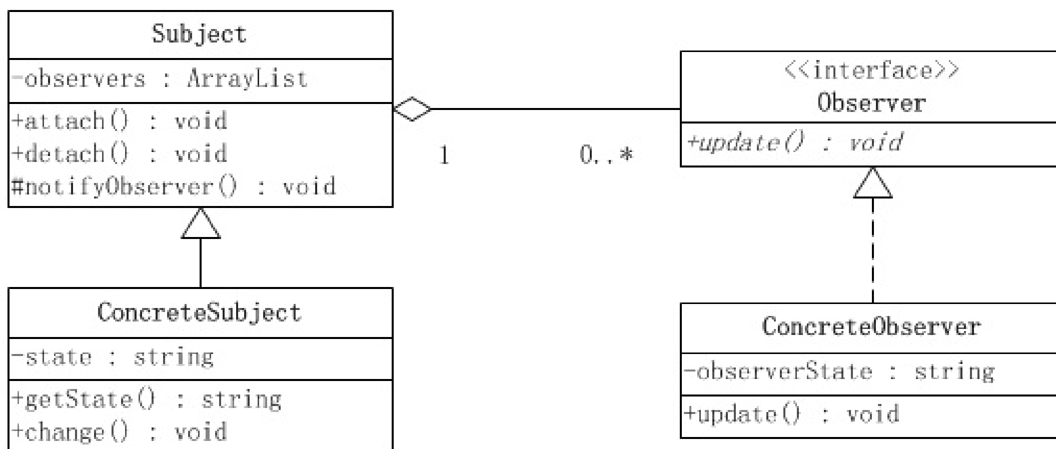
观察者模式定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个主题对象。这个主题对象在状态上发生变化时，会通知所有观察者对象，使它们能够自动更新自己。

二、观察者模式的结构

一个软件系统里面包含了各种对象，就像一片欣欣向荣的森林充满了各种生物一样。在一片森林中，各种生物彼此依赖和约束，形成一个个生物链。一种生物的状态变化会造成其他一些生物的相应行动，每一个生物都处于别的生物的互动之中。

同样，一个软件系统常常要求在某一个对象的状态发生变化的时候，某些其他的对象做出相应的改变。做到这一点的设计方案有很多，但是为了使系统能够易于复用，应该选择低耦合度的设计方案。减少对象之间的耦合有利于系统的复用，但是同时设计师需要使这些低耦合度的对象之间能够维持行动的协调一致，保证高度的协作。观察者模式是满足这一要求的各种设计方案中最重要的一种。

下面以一个简单的示意性实现为例，讨论观察者模式的结构。



观察者模式所涉及的角色有：

- 抽象主题**(Subject)**角色：抽象主题角色把所有对观察者对象的引用保存在一个聚集（比如ArrayList对象）里，每个主题都可以有任何数量的观察者。抽象主题提供一个接口，可以增加和删除观察者对象，抽象主题角色又叫做抽象被观察

者(Observable)角色。

● 具体主题**(ConcreteSubject)**角色：将有关状态存入具体观察者对象；在具体主题的内部状态改变时，给所有登记过的观察者发出通知。具体主题角色又叫做具体被观察者(Concrete Observable)角色。

● 抽象观察者**(Observer)**角色：为所有的具体观察者定义一个接口，在得到主题的通知时更新自己，这个接口叫做更新接口。

● 具体观察者**(ConcreteObserver)**角色：存储与主题的状态自恰的状态。具体观察者角色实现抽象观察者角色所要求的更新接口，以便使本身的状态与主题的状态相协调。如果需要，具体观察者角色可以保持一个指向具体主题对象的引用。

三、观察者模式的代码实现

抽象主题角色类(此处是抽象类，改成接口将会提高复用)

```
public abstract class Subject {
    /**
     * 用来保存注册的观察者对象
     */
    private    List
<
Observer
>
 list = new ArrayList
<
Observer
>
();
    /**
     * 注册观察者对象
     * @param observer    观察者对象
     */
    public void attach(Observer observer){

        list.add(observer);
        System.out.println("Attached an observer");
```

```
    }
    /**
     * 删除观察者对象
     * @param observer    观察者对象
     */
    public void detach(Observer observer){

        list.remove(observer);
    }
    /**
     * 通知所有注册的观察者对象
     */
    public void nodifyObservers(String newState){

        for(Observer observer : list){
            observer.update(newState);
        }
    }
}
```

具体主题角色类

```
public class ConcreteSubject extends Subject{

    private String state;

    public String getState() {
        return state;
    }

    public void change(String newState){
        state = newState;
        System.out.println("主题状态为：" + state);
        //状态发生改变，通知各个观察者
        this.nodifyObservers(state);
    }
}
```

抽象观察者角色类

```
public interface Observer {
    /**
     * 更新接口
     * @param state    更新的状态
     */
    public void update(String state);
}
```

具体观察者角色类

```
public class ConcreteObserver implements Observer {
    //观察者的状态
    private String observerState;

    @Override
    public void update(String state) {
        /**
         * 更新观察者的状态，使其与目标的状态保持一致
         */
        observerState = state;
        System.out.println("状态为："+observerState);
    }
}
```

具体使用：

```java
public class Client {

    public static void main(String[] args) {
        //创建主题对象
        ConcreteSubject subject = new ConcreteSubject();
        //创建观察者对象
        Observer observer = new ConcreteObserver();
        //将观察者对象登记到主题对象上
        subject.attach(observer);
        //改变主题对象的状态
        subject.change("new state");
    }
}
```

当主题对象的状态改变时，将通知所有观察者，观察者接收到主题对象的通知后，将可以进行其他操作，进行响应。

四、推模型和拉模型

在观察者模式中，又分为推模型和拉模型两种方式。

- 推模型

　　主题对象向观察者推送主题的详细信息，不管观察者是否需要，推送的信息通常是主题对象的全部或部分数据。

- 拉模型

　　主题对象在通知观察者的时候，只传递少量信息。如果观察者需要更具体的信息，由观察者主动到主题对象中获取，相当于是观察者从主题对象中拉数据。一般这种模型的实现中，会把主题对象自身通过update()方法传递给观察者，这样在观察者需要获取数据的时候，就可以通过这个引用来获取了。

根据上面的描述，发现前面的例子就是典型的推模型，下面给出一个拉模型的实例。

拉模型的抽象观察者类

拉模型通常都是把主题对象当做参数传递。

```
public interface Observer {
    /**
     * 更新接口
     * @param subject 传入主题对象，方面获取相应的主题对象的状态
     */
    public void update(Subject subject);
}
```

拉模型的具体观察者类

```
public class ConcreteObserver implements Observer {
    //观察者的状态
    private String observerState;

    @Override
    public void update(Subject subject) {
        /**
         * 更新观察者的状态，使其与目标的状态保持一致
         */
        observerState = ((ConcreteSubject)subject).getState();
        System.out.println("观察者状态为："+observerState);
    }

}
```

拉模型的抽象主题类

拉模型的抽象主题类主要的改变是nodifyObservers()方法。在循环通知观察者的时候，也就是循环调用观察者的update()方法的时候，传入的参数不同了。

```
public abstract class Subject {
    /**
     * 用来保存注册的观察者对象
     */
    private    List
<
Observer
>
 list = new ArrayList
```

```
<
Observer
>
();
    /**
     * 注册观察者对象
     * @param observer    观察者对象
     */
    public void attach(Observer observer){

        list.add(observer);
        System.out.println("Attached an observer");
    }
    /**
     * 删除观察者对象
     * @param observer    观察者对象
     */
    public void detach(Observer observer){

        list.remove(observer);
    }
    /**
     * 通知所有注册的观察者对象
     */
    public void nodifyObservers(){

        for(Observer observer : list){
            observer.update(this);
        }
    }
}
```

拉模型的具体主题类

跟推模型相比，有一点变化，就是调用通知观察者的方法的时候，不需要传入参数了。

```
public class ConcreteSubject extends Subject{

    private String state;

    public String getState() {
        return state;
    }

    public void change(String newState){
        state = newState;
        System.out.println("主题状态为：" + state);
        //状态发生改变，通知各个观察者
        this.nodifyObservers();
    }
}
```

两种模式的比较

- 推模型是假定主题对象知道观察者需要的数据；而拉模型是主题对象不知道观察者具体需要什么数据，没有办法的情况下，干脆把自身传递给观察者，让观察者自己去按需要取值。

- 推模型可能会使得观察者对象难以复用，因为观察者的update()方法是按需要定义的参数，可能无法兼顾没有考虑到的使用情况。这就意味着出现新情况的时候，就可能提供新的update()方法，或者是干脆重新实现观察者；而拉模型就不会造成这样的情况，因为拉模型下，update()方法的参数是主题对象本身，这基本上是主题对象能传递的最大数据集合了，基本上可以适应各种情况的需要。

五、JAVA提供的对观察者模式的支持

在JAVA语言的java.util库里面，提供了一个Observable类以及一个Observer接口，构成JAVA语言对观察者模式的支持。

Observer接口

这个接口只定义了一个方法，即update()方法，当被观察者对象的状态发生变化时，被观察者对象的notifyObservers()方法就会调用这一方法。

```
public interface Observer {

    void update(Observable o, Object arg);
}
```

Observable类

被观察者类都是java.util.Observable类的子类。java.util.Observable提供公开的方法支持观察者对象，这些方法中有两个对Observable的子类非常重要：一个是setChanged()，另一个是notifyObservers()。第一方法setChanged()被调用之后会设置一个内部标记变量，代表被观察者对象的状态发生了变化。第二个是notifyObservers()，这个方法被调用时，会调用所有登记过的观察者对象的update()方法，使这些观察者对象可以更新自己。

```
public class Observable {
    private boolean changed = false;
    private Vector obs;

    /** Construct an Observable with zero Observers. */

    public Observable() {
    obs = new Vector();
    }

    /**
     * 将一个观察者添加到观察者聚集上面
     */
    public synchronized void addObserver(Observer o) {
        if (o == null)
            throw new NullPointerException();
    if (!obs.contains(o)) {
        obs.addElement(o);
    }
    }

    /**
     * 将一个观察者从观察者聚集上删除
     */
    public synchronized void deleteObserver(Observer o) {
```

```
        obs.removeElement(o);
    }

    public void notifyObservers() {
    notifyObservers(null);
    }

    /**
     * 如果本对象有变化（那时hasChanged 方法会返回true）
     * 调用本方法通知所有登记的观察者，即调用它们的update()方法
     * 传入this和arg作为参数
     */
    public void notifyObservers(Object arg) {

        Object[] arrLocal;

    synchronized (this) {

        if (!changed)
                return;
            arrLocal = obs.toArray();
            clearChanged();
        }

        for (int i = arrLocal.length-1; i
>
=0; i--)
            ((Observer)arrLocal[i]).update(this, arg);
    }

    /**
     * 将观察者聚集清空
     */
    public synchronized void deleteObservers() {
    obs.removeAllElements();
    }

    /**
     * 将“已变化”设置为true
     */
```

```
    protected synchronized void setChanged() {
    changed = true;
    }

    /**
     * 将“已变化”重置为false
     */
    protected synchronized void clearChanged() {
    changed = false;
    }

    /**
     * 检测本对象是否已变化
     */
    public synchronized boolean hasChanged() {
    return changed;
    }

    /**
     * Returns the number of observers of this
<
tt
>
Observable
<
/tt
>
 object.
     *
     * @return  the number of observers of this object.
     */
    public synchronized int countObservers() {
    return obs.size();
    }
}
```

这个类代表一个被观察者对象，有时称之为主题对象。一个被观察者对象可以有数个观察者对象，每个观察者对象都是实现Observer接口的对象。在被观察者发生变化时，会调用Observable的notifyObservers()方法，此方法调用所有的具体观察者

的update()方法，从而使所有的观察者都被通知更新自己。

使用**JAVA**对观察者模式的支持

这里给出一个非常简单的例子，说明怎样使用JAVA所提供的对观察者模式的支持。在这个例子中，被观察对象叫做Watched；而观察者对象叫做Watcher。Watched对象继承自java.util.Observable类；而Watcher对象实现了java.util.Observer接口。另外有一个Test类扮演客户端角色。

被观察者**Watched**类

```
public class Watched extends Observable{

    private String data = "";

    public String getData() {
        return data;
    }

    public void setData(String data) {

        if(!this.data.equals(data)){
            this.data = data;
            setChanged();
           notifyObservers();
        }
    }


}
```

观察者**Watcher** 类

```
public class Watcher implements Observer{

    public Watcher(Observable o){
        o.addObserver(this);
    }

    @Override
    public void update(Observable o, Object arg) {

        System.out.println("状态发生改变：" + ((Watched)o).getData
());
    }
}
```

具体使用：

```
public static void main(String[] args) {

        //创建被观察者对象
        Watched watched = new Watched();
        //创建观察者对象，并将被观察者对象登记
        Observer watcher = new Watcher(watched);
        //给被观察者状态赋值
        watched.setData("start");
        watched.setData("run");
        watched.setData("stop");

    }
```

总结：

观察者模式一般是一对多的情形，本文中篇幅有限，都是一对一的例子。当一对多时，通过Java内置的观察者模式时，通知多个观察者的顺序不是固定的。所以如果依赖此顺序的话，要自己实现观察者模式。同时Java内置的观察者模式中，Observerable是个类，所以在子类继承了该类后就不能继承其他类，导致复用受到限制，自己实现观察者模式时可以设置为接口，提高复用。

时间： 2018-05-12 01:44:07

一、策略模式的简介

1.定义

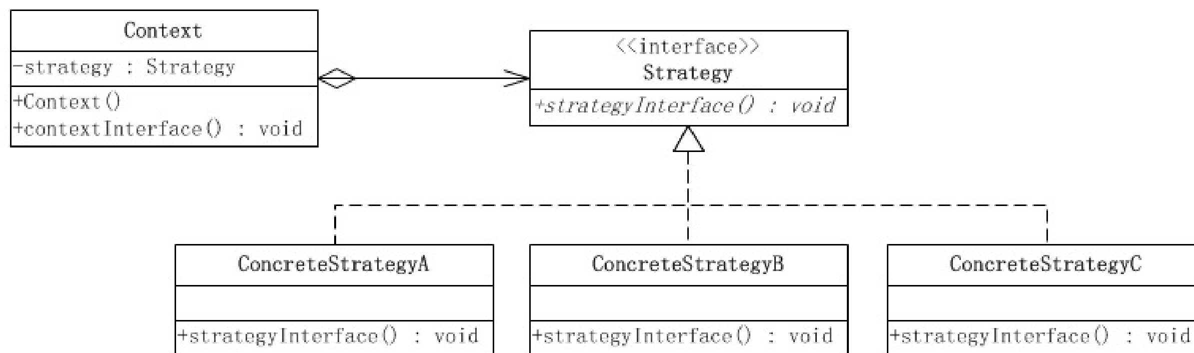
策略模式属于对象的行为模式。其用意是针对一组算法，将每一个算法封装到具有共同接口的独立的类中，从而使得它们可以相互替换。策略模式使得算法可以在不影响到客户端的情况下发生变化。

2.使用场景

针对一个对象，其行为有些是固定的不变的，有些是容易变化的，针对不同情况有不同的表现形式。那么对于这些容易变化的行为，我们不希望将其实现绑定在对象中，而是希望以动态的形式，针对不同情况产生不同的应对策略。那么这个时候就要用到策略模式了。简言之，策略模式就是为了应对对象中复杂多变的行为而产生的。

二、策略模式的结构

策略模式是对算法的包装，是把调用算法的责任（行为）和算法本身（行为实现）分割开来，委派给不同的对象管理。策略模式通常把一个系列的算法包装到一系列的策略类里面，作为一个抽象策略类的子类。用一句话来说，就是："准备一组算法，并将每一个算法封装起来，使得它们可以互换"。下面就以一个示意性的实现讲解策略模式实例的结构。



这个模式涉及到三个角色：

- 环境(Context)角色：持有一个Strategy的引用，即具有复杂多变行为的对象。

- 抽象策略(Strategy)角色：这是一个抽象角色，通常由一个接口或抽象类实现。此角色给出所有的具体策略类所需的接口。

- 具体策略(ConcreteStrategy)角色：包装了相关的算法或行为。

三、具体场景实现

假设现在要设计一个贩卖各类书籍的电子商务网站的购物车系统。一个最简单的情况就是把所有货品的单价乘上数量，但是实际情况肯定比这要复杂。比如，本网站可能对所有的高级会员提供每本20%的促销折扣；对中级会员提供每本10%的促销折扣；对初级会员没有折扣。

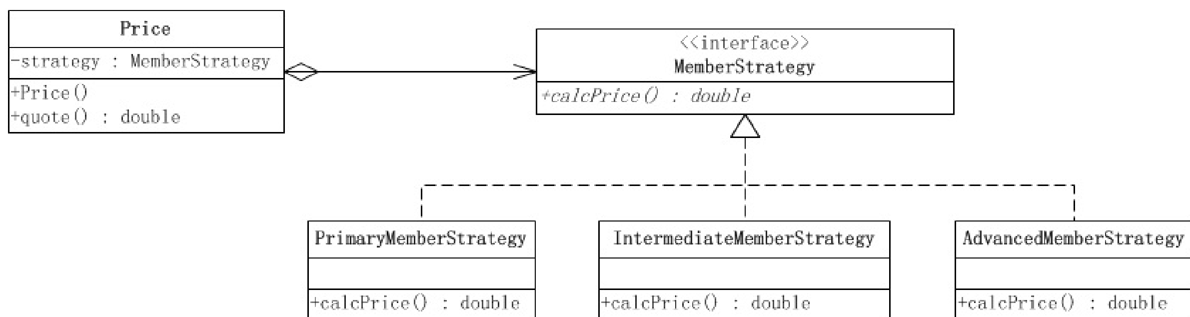
根据描述，折扣是根据以下的几个算法中的一个进行的：

算法一：对初级会员没有折扣。

算法二：对中级会员提供10%的促销折扣。

算法三：对高级会员提供20%的促销折扣。

使用策略模式来实现的结构图如下：



抽象折扣类

```
public interface MemberStrategy {
    /**
     * 计算图书的价格
     * @param booksPrice    图书的原价
     * @return    计算出打折后的价格
     */
    public double calcPrice(double booksPrice);
}
```

初级会员折扣类

```
public class PrimaryMemberStrategy implements MemberStrategy {

    @Override
    public double calcPrice(double booksPrice) {

        System.out.println("对于初级会员的没有折扣");
        return booksPrice;
    }

}
```

中级会员折扣类

```
public class IntermediateMemberStrategy implements MemberStrategy
 {

    @Override
    public double calcPrice(double booksPrice) {

        System.out.println("对于中级会员的折扣为10%");
        return booksPrice * 0.9;
    }

}
```

高级会员折扣类

```
public class AdvancedMemberStrategy implements MemberStrategy {

    @Override
    public double calcPrice(double booksPrice) {

        System.out.println("对于高级会员的折扣为20%");
        return booksPrice * 0.8;
    }
}
```

价格类

```
public class Price {
    //持有一个具体的策略对象
    private MemberStrategy strategy;
    /**
     * 构造函数，传入一个具体的策略对象
     * @param strategy    具体的策略对象
     */
    public Price(MemberStrategy strategy){
        this.strategy = strategy;
    }

    /**
     * 计算图书的价格
     * @param booksPrice    图书的原价
     * @return    计算出打折后的价格
     */
    public double quote(double booksPrice){
        return this.strategy.calcPrice(booksPrice);
    }
}
```

具体调用：

```
    public static void main(String[] args) {
        //选择并创建需要使用的策略对象
        MemberStrategy strategy = new AdvancedMemberStrategy();
        //创建环境
        Price price = new Price(strategy);
        //计算价格
        double quote = price.quote(300);
        System.out.println("图书的最终价格为：" + quote);
    }
```

四、对策略模式的深入认识

策略模式对多态的使用

通过让环境类持有一个抽象策略类（超类）的引用，在生成环境类实例对象时，让该引用指向具体的策略子类。再对应的方法调用中，就会通过Java的多态，调用对应策略子类的方法。从而可以相互替换，不需要修改环境类内部的实现。同时，在有新的需求的情况下，也只需要修改策略类即可，降低与环境类之间的耦合度。

策略模式的重心

策略模式的重心不是如何实现算法，而是如何组织、调用这些算法，从而让程序结构更灵活，具有更好的维护性和扩展性。

算法的平等性

策略模式一个很大的特点就是各个策略算法的平等性。对于一系列具体的策略算法，大家的地位是完全一样的，正因为这个平等性，才能实现算法之间可以相互替换。所有的策略算法在实现上也是相互独立的，相互之间是没有依赖的。

所以可以这样描述这一系列策略算法：策略算法是相同行为的不同实现。

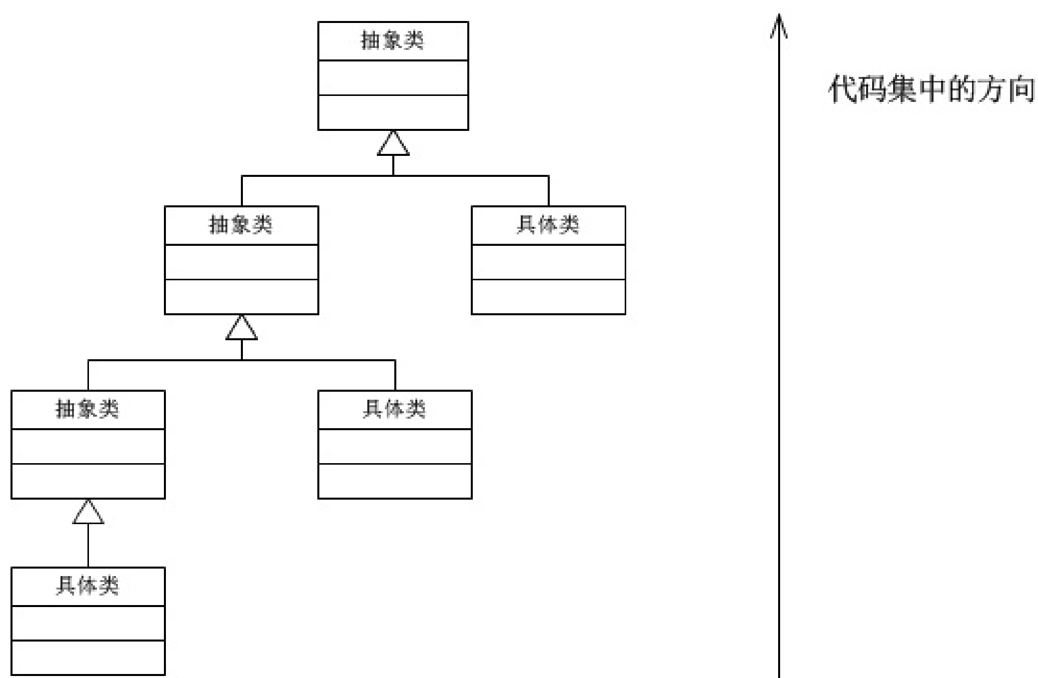
运行时策略的唯一性

运行期间，策略模式在每一个时刻只能使用一个具体的策略实现对象，虽然可以动态地在不同的策略实现中切换，但是同时只能使用一个。

公有的行为

经常见到的是，所有的具体策略类都有一些公有的行为。这时候，就应当把这些公有的行为放到共同的抽象策略角色Strategy类里面。当然这时候抽象策略角色必须要用Java抽象类实现，而不能使用接口。

这其实也是典型的将代码向继承等级结构的上方集中的标准做法。



五、策略模式的优缺点

策略模式的优点

（1）策略模式提供了管理相关的算法族的办法。策略类的等级结构定义了一个算法或行为族。恰当使用继承可以把公共的代码移到父类里面，从而避免代码重复。

（2）使用策略模式可以避免使用多重条件(if-else)语句。多重条件语句不易维护，它把采取哪一种算法或采取哪一种行为的逻辑与算法或行为的逻辑混合在一起，统统列在一个多重条件语句里面，比使用继承的办法还要原始和落后。

策略模式的缺点

（1）客户端必须知道所有的策略类，并自行决定使用哪一个策略类。这就意味着客户端必须理解这些算法的区别，以便适时选择恰当的算法类。换言之，策略模式只适用于客户端知道算法或行为的情况。

（2）由于策略模式把每个具体的策略实现都单独封装成为类，如果备选的策略很多的话，那么对象的数目就会很可观。

一、模版方法模式的定义

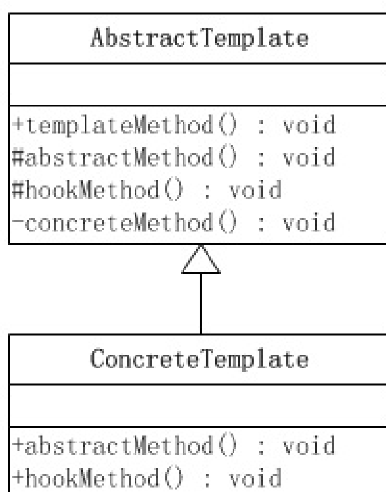
模板方法模式是类的行为模式。准备一个抽象类，将部分逻辑以具体方法以及具体构造函数的形式实现，然后声明一些抽象方法来迫使子类实现剩余的逻辑。不同的子类可以以不同的方式实现这些抽象方法，从而对剩余的逻辑有不同的实现。这就是模板方法模式的用意。

二、模版方法模式的结构

模板方法模式是所有模式中最为常见的几个模式之一，是基于继承的代码复用的基本技术。

模板方法模式需要开发抽象类和具体子类的设计师之间的协作。一个设计师负责给出一个算法的轮廓和骨架，另一些设计师则负责给出这个算法的各个逻辑步骤。代表这些具体逻辑步骤的方法称做基本方法(primitive method)；而将这些基本方法汇总起来的方法叫做模板方法(template method)，这个设计模式的名字就是从此而来。

模板方法所代表的行为称为顶级行为，其逻辑称为顶级逻辑。模板方法模式的静态结构图如下所示：



这里涉及到两个角色：

抽象模板**(Abstract Template)**角色有如下责任：

■ 定义了一个或多个抽象操作，以便让子类实现。这些抽象操作叫做基本操作，它们是一个顶级逻辑的组成步骤。

■　定义并实现了一个模板方法。这个模板方法一般是一个具体方法，它给出了一个顶级逻辑的骨架，而逻辑的组成步骤在相应的抽象操作中，推迟到子类实现。顶级逻辑也有可能调用一些具体方法。

具体模板(**Concrete Template**)角色又如下责任：

■　实现父类所定义的一个或多个抽象方法，它们是一个顶级逻辑的组成步骤。

■　每一个抽象模板角色都可以有任意多个具体模板角色与之对应，而每一个具体模板角色都可以给出这些抽象方法（也就是顶级逻辑的组成步骤）的不同实现，从而使得顶级逻辑的实现各不相同。

抽象模板角色类，**abstractMethod()**、**hookMethod()**等基本方法是顶级逻辑的组成步骤，这个顶级逻辑由**templateMethod()**方法代表。

```
public abstract class AbstractTemplate {
    /**
     * 模板方法
     */
    public void templateMethod(){
        //调用基本方法
        abstractMethod();
        hookMethod();
        concreteMethod();
    }
    /**
     * 基本方法的声明（由子类实现）
     */
    protected abstract void abstractMethod();
    /**
     * 基本方法(空方法)
     */
    protected void hookMethod(){}
    /**
     * 基本方法（已经实现）
     */
    private final void concreteMethod(){
        //业务相关的代码
    }
}
```

具体模板角色类，实现了父类所声明的基本方法，**abstractMethod()**方法所代表的就是强制子类实现的剩余逻辑，而**hookMethod()**方法是可选择实现的逻辑，不是必须实现的。

```
public class ConcreteTemplate extends AbstractTemplate{
    //基本方法的实现
    @Override
    public void abstractMethod() {
        //业务相关的代码
    }
    //重写父类的方法
    @Override
    public void hookMethod() {
        //业务相关的代码
    }
}
```

模板模式的关键是：子类可以置换掉父类的可变部分，但是子类却不可以改变模板方法所代表的顶级逻辑。

每当定义一个新的子类时，不要按照控制流程的思路去想，而应当按照“责任”的思路去想。换言之，应当考虑哪些操作是必须置换掉的，哪些操作是可以置换掉的，以及哪些操作是不可以置换掉的。使用模板模式可以使这些责任变得清晰。

三、模板方法模式中的方法

模板方法中的方法可以分为两大类：模板方法和基本方法。

模板方法

一个模板方法是定义在抽象类中的，把基本操作方法组合在一起形成一个总算法或一个总行为的方法。

一个抽象类可以有任意多个模板方法，而不限于一个。每一个模板方法都可以调用任意多个具体方法。

基本方法

基本方法又可以分为三种：抽象方法(Abstract Method)、具体方法(Concrete Method)和钩子方法(Hook Method)。

- 抽象方法：一个抽象方法由抽象类声明，由具体子类实现。在Java语言里抽象方法以abstract关键字标示。
- 具体方法：一个具体方法由抽象类声明并实现，而子类并不实现或置换。
- 钩子方法：一个钩子方法由抽象类声明并实现，而子类会加以扩展。通常抽象类给出的实现是一个空实现，作为方法的默认实现。

在上面的例子中，AbstractTemplate是一个抽象类，它带有三个方法。其中abstractMethod()是一个抽象方法，它由抽象类声明为抽象方法，并由子类实现；hookMethod()是一个钩子方法，它由抽象类声明并提供默认实现，并且由子类置换掉。concreteMethod()是一个具体方法，它由抽象类声明并实现。

默认钩子方法

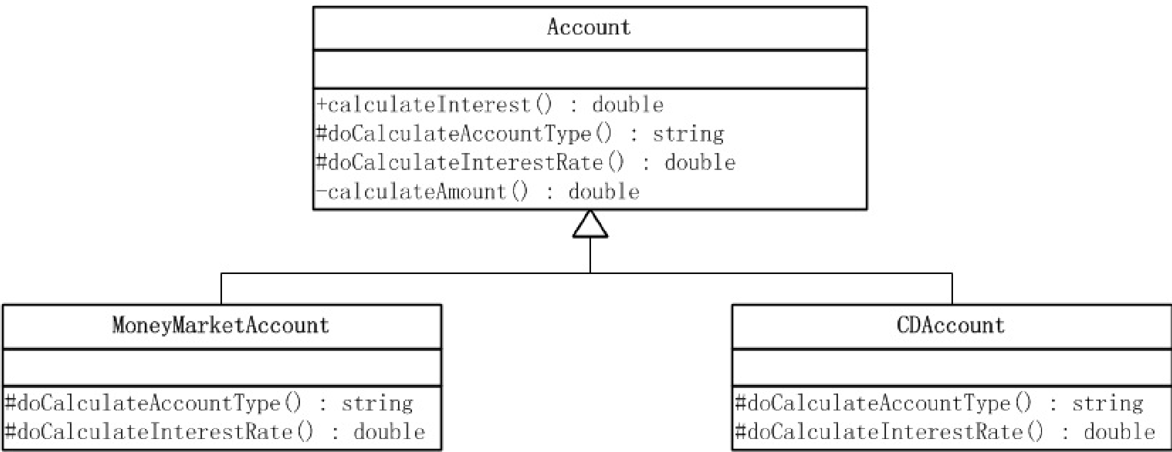
一个钩子方法常常由抽象类给出一个空实现作为此方法的默认实现。这种空的钩子方法叫做“Do Nothing Hook”。具体模版类中可以选择是否重写钩子方法，通常重写钩子方法是为了对模版方法中的步骤进行控制，判断钩子方法中的状态，是否进行下一步操作。

四、模版方法的具体实例

考虑一个计算存款利息的例子。假设系统需要支持两种存款账号，即货币市场(Money Market)账号和定期存款(Certificate of Deposite)账号。这两种账号的存款利息是不同的，因此，在计算一个存户的存款利息额时，必须区分两种不同的账号类型。

这个系统的总行为应当是计算出利息，这也就决定了作为一个模板方法模式的顶级逻辑应当是利息计算。由于利息计算涉及到两个步骤：一个基本方法给出账号种类，另一个基本方法给出利息百分比。这两个基本方法构成具体逻辑，因为账号的类型不同，所以具体逻辑会有所不同。

显然，系统需要一个抽象角色给出顶级行为的实现，而将两个作为细节步骤的基本方法留给具体子类实现。由于需要考虑的账号有两种：一是货币市场账号，二是定期存款账号。系统的类结构如下图所示。


Account
+calculateInterest() : double
#doCalculateAccountType() : string
#doCalculateInterestRate() : double
-calculateAmount() : double
MoneyMarketAccount
#doCalculateAccountType() : string
#doCalculateInterestRate() : double
CDAccount
#doCalculateAccountType() : string
#doCalculateInterestRate() : double

抽象模板角色类

```
public abstract class Account {
    /**
     * 模板方法，计算利息数额
     * @return    返回利息数额
     */
    public final double calculateInterest(){
        double interestRate = doCalculateInterestRate();
        String accountType = doCalculateAccountType();
        double amount = calculateAmount(accountType);
        return amount * interestRate;
    }
    /**
     * 基本方法留给子类实现
     */
    protected abstract String doCalculateAccountType();
    /**
     * 基本方法留给子类实现
     */
    protected abstract double doCalculateInterestRate();
    /**
     * 基本方法，已经实现
     */
    private double calculateAmount(String accountType){
        /**
         * 省略相关的业务逻辑
         */
        return 7243.00;
    }
}
```

具体模板角色类

```
public class MoneyMarketAccount extends Account {

    @Override
    protected String doCalculateAccountType() {

        return "Money Market";
    }

    @Override
    protected double doCalculateInterestRate() {

        return 0.045;
    }

}
```

```
public class CDAccount extends Account {

    @Override
    protected String doCalculateAccountType() {
        return "Certificate of Deposite";
    }

    @Override
    protected double doCalculateInterestRate() {
        return 0.06;
    }

}
```

客户端类

```
public class Client {

    public static void main(String[] args) {
        Account account = new MoneyMarketAccount();
        System.out.println("货币市场账号的利息数额为：" + account.ca
lculateInterest());
        account = new CDAccount();
        System.out.println("定期账号的利息数额为：" + account.calcul
ateInterest());
    }

}
```

五、模板方法模式效果与适用场景

模板方法模式是基于继承的代码复用技术，它体现了面向对象的诸多重要思想，是一种使用较为频繁的模式。模板方法模式广泛应用于框架设计中，以确保通过父类来控制处理流程的逻辑顺序（如框架的初始化，测试流程的设置等）。

在以下情况下可以考虑使用模板方法模式：

(1) 对一些复杂的算法进行分割，将其算法中固定不变的部分设计为模板方法和父类具体方法，而一些可以改变的细节由其子类来实现。即：一次性实现一个算法的不变部分，并将可变的行为留给子类来实现。

(2) 各子类中公共的行为应被提取出来并集中到一个公共父类中以避免代码重复。

(3) 需要通过子类来决定父类算法中某个步骤是否执行，实现子类对父类的反向控制。

六、模版方法模式的优缺点

优点

(1) 在父类中形式化地定义一个算法，而由它的子类来实现细节的处理，在子类实现详细的处理算法时并不会改变算法中步骤的执行次序。

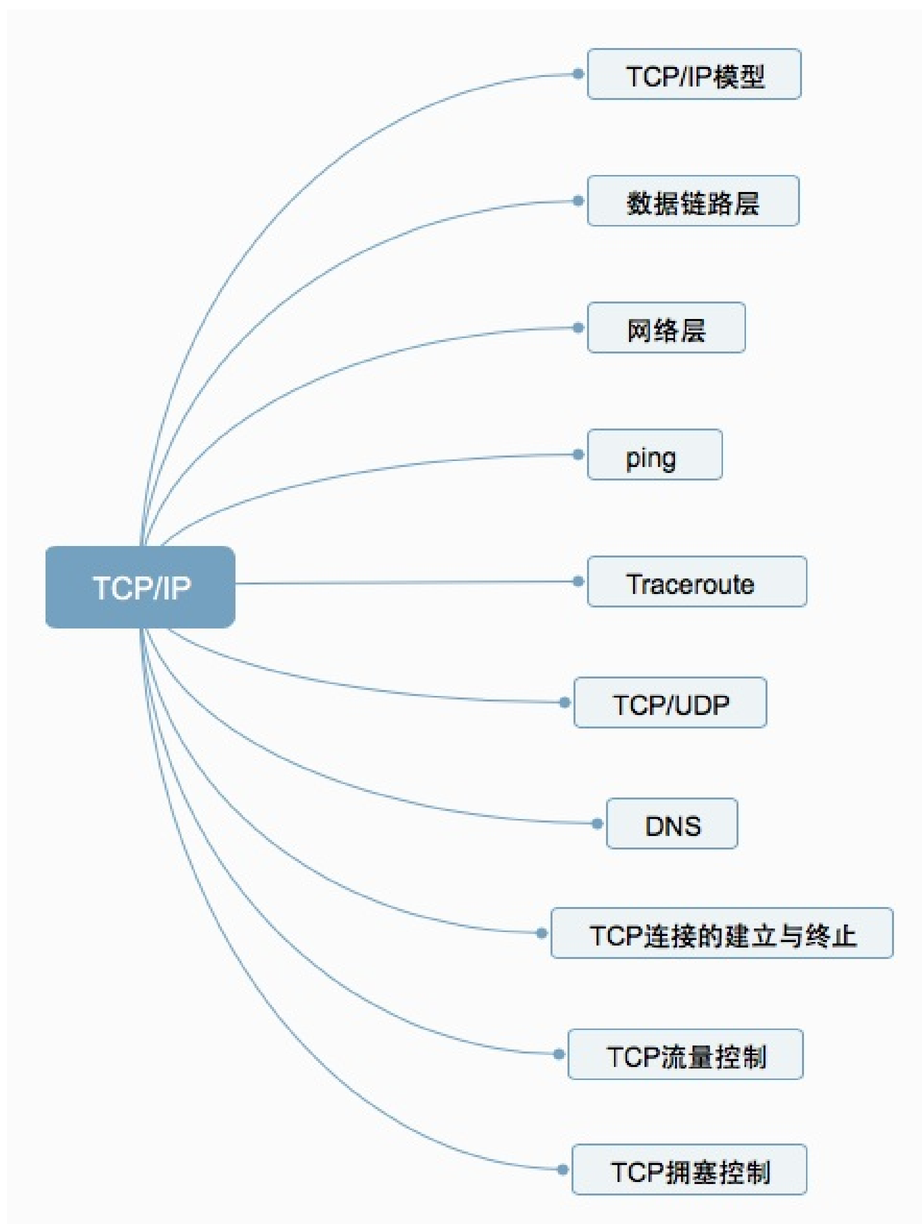
(2) 模板方法模式是一种代码复用技术，它在类库设计中尤为重要，它提取了类库中的公共行为，将公共行为放在父类中，而通过其子类来实现不同的行为，它鼓励我们恰当使用继承来实现代码复用。

(3) 可实现一种反向控制结构，通过子类覆盖父类的钩子方法来决定某一特定步骤是否需要执行。

(4) 在模板方法模式中可以通过子类来覆盖父类的基本方法，不同的子类可以提供基本方法的不同实现，更换和增加新的子类很方便，符合单一职责原则和开闭原则。

缺点

需要为每一个基本方法的不同实现提供一个子类，如果父类中可变的基本方法太多，将会导致类的个数增加，系统更加庞大，设计也更加抽象，此时，可结合桥接模式来进行设计。



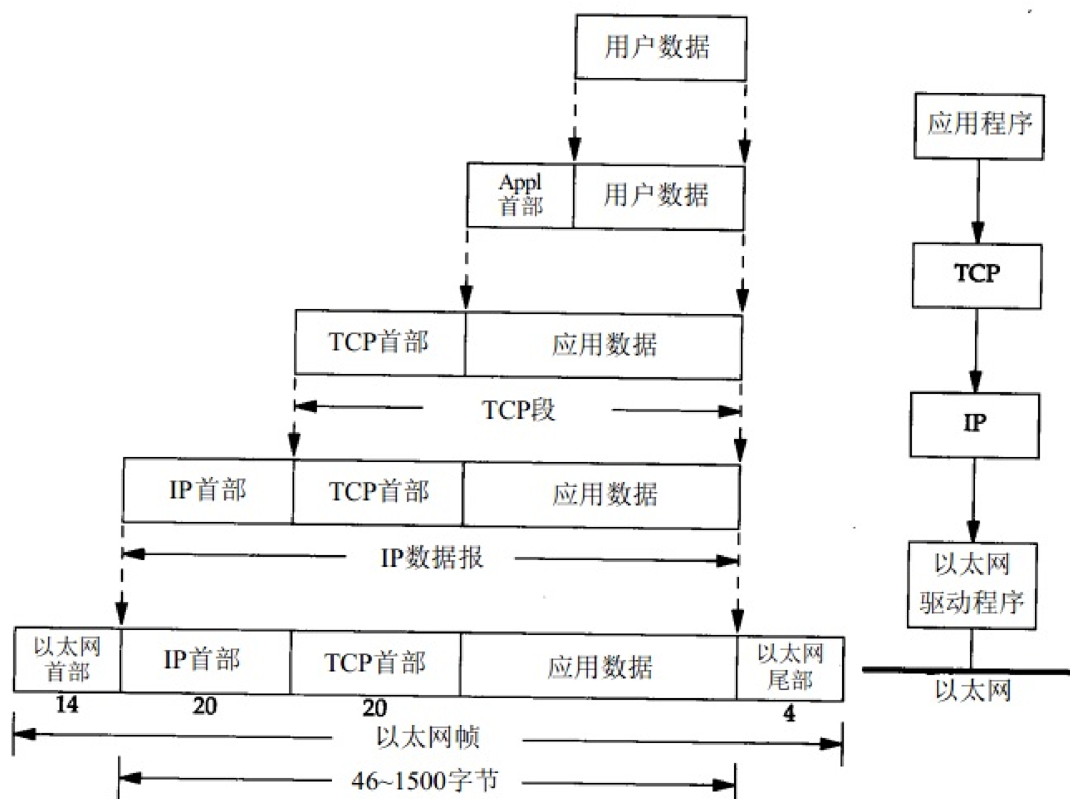
一、TCP/IP模型

TCP/IP协议模型（Transmission Control Protocol/Internet Protocol），包含了一系列构成互联网基础的网络协议，是Internet的核心协议。

基于TCP/IP的参考模型将协议分成四个层次，它们分别是链路层、网络层、传输层和应用层。下图表示TCP/IP模型与OSI模型各层的对照关系。

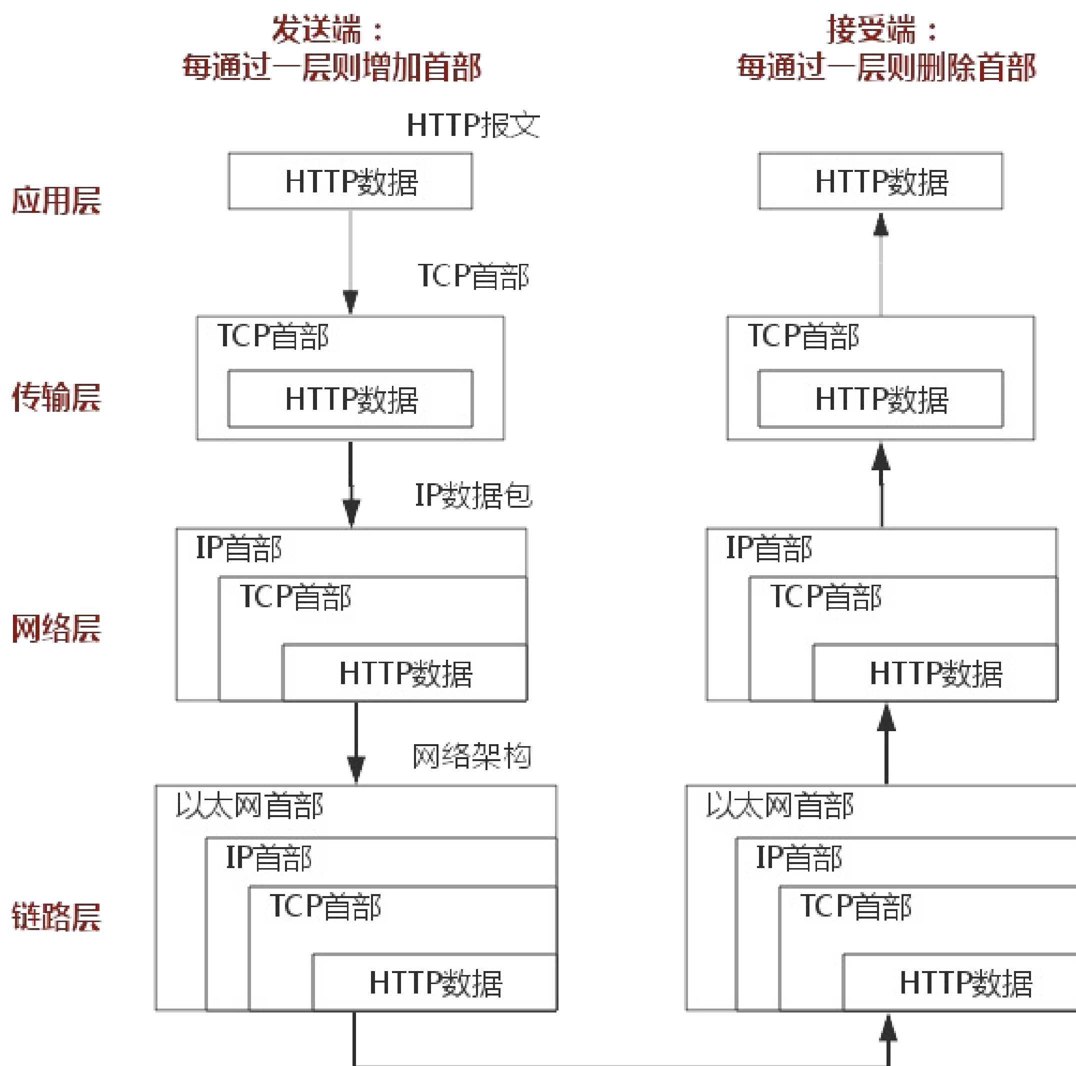


TCP/IP协议族按照层次由上到下，层层包装。最上面的是应用层，这里面有http，ftp，等等我们熟悉的协议。而第二层则是传输层，著名的TCP和UDP协议就在这个层次。第三层是网络层，IP协议就在这里，它负责对数据加上IP地址和其他的数据以确定传输的目标。第四层是数据链路层，这个层次为待传送的数据加入一个以太网协议头，并进行CRC编码，为最后的数据传输做准备。



数据进入协议栈时的封装过程

上图清楚地表示了TCP/IP协议中每个层的作用，而TCP/IP协议通信的过程其实就对应着数据入栈与出栈的过程。入栈的过程，数据发送方每层不断地封装首部与尾部，添加一些传输的信息，确保能传输到目的地。出栈的过程，数据接收方每层不断地拆除首部与尾部，得到最终传输的数据。



上图以HTTP协议为例，具体说明。

二、数据链路层

物理层负责0、1比特流与物理设备电压高低、光的闪灭之间的互换。数据链路层负责将0、1序列划分为数据帧从一个节点传输到临近的另一个节点，这些节点是通过MAC来唯一标识的(MAC，物理地址，一个主机会有一个MAC地址)。



- 封装成帧：把网络层数据报加头和尾，封装成帧，帧头中包括源MAC地址和目的MAC地址。
- 透明传输：零比特填充、转义字符。
- 可靠传输：在出错率很低的链路上很少用，但是无线链路WLAN会保证可靠传输。
- 差错检测(CRC)：接收者检测错误,如果发现差错，丢弃该帧。

三、网络层

1.IP协议

IP协议是TCP/IP协议的核心，所有的TCP，UDP，ICMP，IGCP的数据都以IP数据格式传输。要注意的是，IP不是可靠的协议，这是说，IP协议没有提供一种数据未传达以后的处理机制，这被认为是上层协议——TCP或UDP要做的事情。

1.1 IP地址

在数据链路层中我们一般通过MAC地址来识别不同的节点，而在IP层我们也要有一个类似的地址标识，这就是IP地址。

32位IP地址分为网络位和地址位，这样做可以减少路由器中路由表记录的数目，有了网络地址，就可以限定拥有相同网络地址的终端都在同一个范围内，那么路由表只需要维护一条这个网络地址的方向，就可以找到相应的这些终端了。

- A类IP地址：1.0.0.0~127.0.0.0
- B类IP地址：128.0.0.0~191.255.255.255
- C类IP地址：192.0.0.0~223.255.255.255

1.2 IP协议头



这里只介绍：八位的TTL（Time To Live）字段。这个字段规定该数据包在穿过多少个路由之后才会被抛弃。某个IP数据包每穿过一个路由器，该数据包的TTL数值就会减少1，当该数据包的TTL成为零，它就会被自动抛弃。

这个字段的最大值也就是255，也就是说一个协议包也就在路由器里面穿行255次就会被抛弃了，根据系统的不同，这个数字也不一样，一般是32或者是64。

2.ARP及RARP协议

ARP 是根据IP地址获取MAC地址的一种协议。

ARP（地址解析）协议是一种解析协议，本来主机是完全不知道这个IP对应的是哪个主机的哪个接口，当主机要发送一个IP包的时候，会首先查一下自己的ARP高速缓存（就是一个IP-MAC地址对应表缓存）。

如果查询的IP-MAC值对不存在，那么主机就向网络发送一个ARP协议广播包，这个广播包里面就有待查询的IP地址，而直接收到这份广播的包的所有主机都会查询自己的IP地址。如果收到广播包的某一个主机发现自己符合条件，那么就准备好一个包含自己的MAC地址的ARP包传送给发送ARP广播的主机。

而广播主机拿到ARP包后会更新自己的ARP缓存（就是存放IP-MAC对应表的地方）。发送广播的主机就会用新的ARP缓存数据准备好数据链路层的的数据包发送工作。

RARP协议的工作与此相反，不做赘述。

3. ICMP协议

IP协议并不是一个可靠的协议，它不保证数据被送达，那么，自然的，保证数据送达的工作应该由其他的模块来完成。其中一个重要的模块就是ICMP(网络控制报文)协议。ICMP不是高层协议，而是IP层的协议。

当传送IP数据包发生错误。比如主机不可达，路由不可达等等，ICMP协议将会把错误信息封包，然后传送回给主机。给主机一个处理错误的机会，这 也就是为什么说建立在IP层以上的协议是可能做到安全的原因。

四、ping

ping可以说是ICMP的最著名的应用，是TCP/IP协议的一部分。利用“ping”命令可以检查网络是否连通，可以很好地帮助我们分析和判定网络故障。

例如：当我们某一个网站上不去的时候。通常会ping一下这个网站。ping会回显出一些有用的信息。一般的信息如下:

```
Reply from 10.4.24.1: bytes=32 time<1ms TTL=255
Reply from 10.4.24.1: bytes=32 time<1ms TTL=255
Reply from 10.4.24.1: bytes=32 time<1ms TTL=255
Reply from 10.4.24.1: bytes=32 time<1ms TTL=255

Ping statistics for 10.4.24.1:
   Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
   Minimum = 0ms, Maximum = 0ms, Average = 0ms
```

ping这个单词源自声纳定位，而这个程序的作用也确实如此，它利用ICMP协议包来侦测另一个主机是否可达。原理是用类型码为0的ICMP发请 求，受到请求的主机则用类型码为8的ICMP回应。

ping程序来计算间隔时间，并计算有多少个包被送达。用户就可以判断网络大致的情况。我们可以看到， ping给出来了传送的时间和TTL的数据。

五、Traceroute

Traceroute是用来侦测主机到目的主机之间所经路由情况的重要工具，也是最便利的工具。

Traceroute的原理非常非常有意思。它收到到目的主机的IP后，首先给目的主机发送一个TTL=1（Time To Live）的UDP数据包，而经过的第一个路由器收到这个数据包以后，就自动把TTL减1，而TTL变为0以后，路由器就把这个包给抛弃了，并同时产生一个主机不可达的ICMP数据报给主机。主机收到这个数据报以后再发一个TTL=2的UDP数据报给目的主机，然后刺激第二个路由器给主机发ICMP数据报。如此往复直到到达目的主机。这样，traceroute就拿到了所有的路由器IP。

```
C:\Users\Jack>tracert www.baidu.com

通过最多 30 个跃点跟踪
到 www.a.shifen.com [119.75.218.70] 的路由:

  1    <1 毫秒   <1 毫秒   <1 毫秒 59.64.156.1
  2     1 ms      1 ms      1 ms  10.1.1.1
  3    <1 毫秒    1 ms     <1 毫秒 10.0.11.1
  4    <1 毫秒   <1 毫秒   <1 毫秒 10.0.0.1
  5    10 ms      9 ms     10 ms  202.112.42.1
  6    10 ms     10 ms     10 ms  202.112.6.54
  7    10 ms     10 ms     11 ms  192.168.0.5
  8    11 ms     10 ms     11 ms  10.65.190.131
  9    12 ms     10 ms     10 ms  119.75.218.70
```

六、TCP/UDP

TCP/UDP都是是传输层协议，但是两者具有不同的特性，同时也具有不同的应用场景，下面以图表的形式对比分析。

	TCP	UDP
可靠性	可靠	不可靠
连接性	面向连接	无连接
报文	面向字节流	面向报文
效率	传输效率低	传输效率高
双工性	全双工	一对一、一对多、多对一、多对多
流量控制	滑动窗口	无
拥塞控制	慢开始、拥塞避免、快重传、快恢复	无
传输速度	慢	快
应用场景	对效率要求低，对准确性要求高或者要求有连接的场景	对效率要求高，对准确性要求低

面向报文

面向报文的传输方式是应用层交给UDP多长的报文，UDP发送多长的报文，即一次发送一个报文。因此，应用程序必须选择合适大小的报文。若报文太长，则IP层需要分片，降低效率；若太短，会使IP数据报太小。

面向字节流

面向字节流的话，虽然应用程序和TCP的交互是一次一个数据块（大小不等），但TCP把应用程序看成是一连串的无结构的字节流。TCP有一个缓冲，当应用程序传送的数据块太长，TCP就可以把它划分短一些再传送。

关于拥塞控制，流量控制，是TCP的重点，后面讲解。

TCP和UDP协议的一些应用

应用层协议	应用	传输层协议
SMTP	电子邮件	TCP
TELNET	远程终端接入	
HTTP	万维网	
FTP	文件传输	
DNS	域名转换	UDP
TFTP	文件传输	
SNMP	网络管理	
NFS	远程文件服务器	

什么时候应该使用TCP？

当对网络通讯质量有要求的时候，比如：整个数据要准确无误的传递给对方，这往往用于一些要求可靠的应用，比如HTTP、HTTPS、FTP等传输文件的协议，POP、SMTP等邮件传输的协议。

什么时候应该使用UDP？

当对网络通讯质量要求不高的时候，要求网络通讯速度能尽量的快，这时就可以使用UDP。

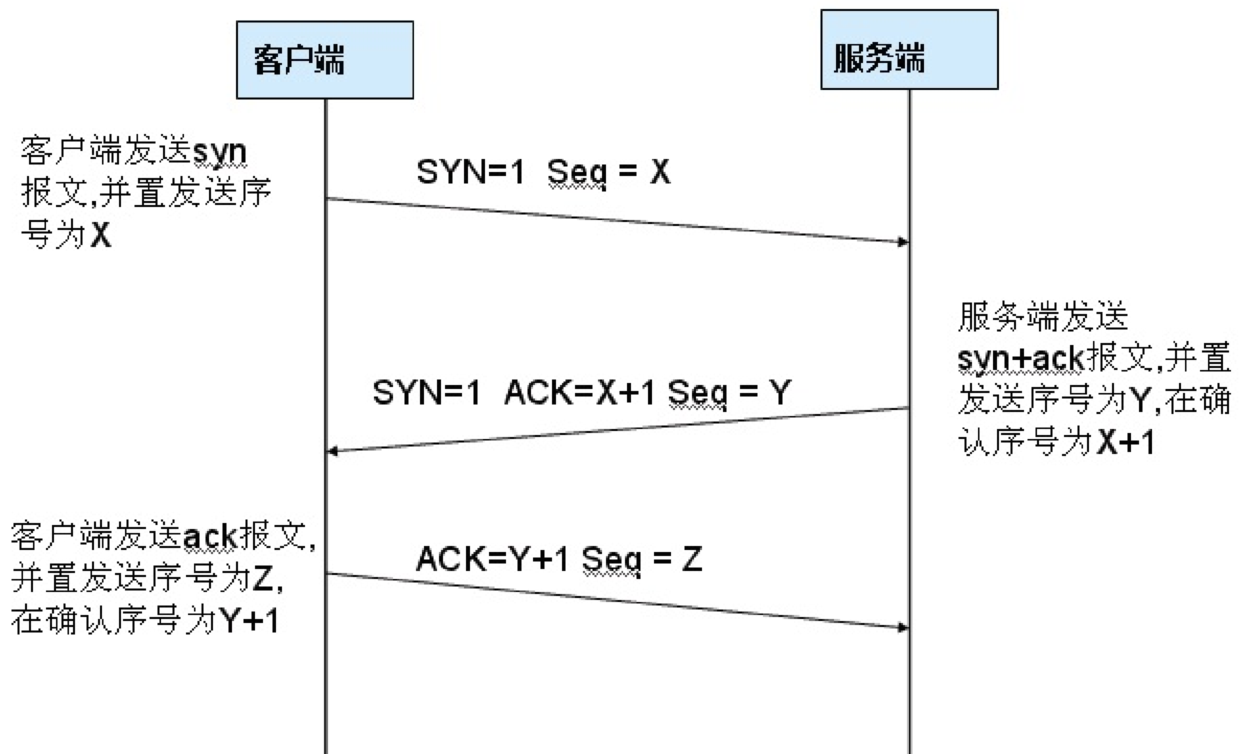
七、DNS

DNS（Domain Name System，域名系统），因特网上作为域名和IP地址相互映射的一个分布式数据库，能够使用户更方便的访问互联网，而不用去记住能够被机器直接读取的IP数串。通过主机名，最终得到该主机名对应的IP地址的过程叫做域名解析（或主机名解析）。DNS协议运行在UDP协议之上，使用端口号53。

八、TCP连接的建立与终止

1.三次握手

TCP是面向连接的，无论哪一方向另一方发送数据之前，都必须先在双方之间建立一条连接。在TCP/IP协议中，TCP协议提供可靠的连接服务，连接是通过三次握手进行初始化的。三次握手的目的是同步连接双方的序列号和确认号并交换 TCP窗口大小信息。



回合	说明
第一次握手	建立连接。客户端发送连接请求报文段，将SYN位置为1，Sequence Number为x；然后，客户端进入SYN_SEND状态，等待服务器的确认；
第二次握手	服务器收到SYN报文段。服务器收到客户端的SYN报文段，需要对这个SYN报文段进行确认，设置Acknowledgment Number为x + 1(Sequence Number + 1)；同时，自己自己还要发送SYN请求信息，将SYN位置为1，Sequence Number为y；服务器端将上述所有信息放到一个报文段（即SYN+ACK报文段）中，一并发送给客户端，此时服务器进入SYN_RECV状态；
第三次握手	客户端收到服务器的SYN+ACK报文段。然后将Acknowledgment Number设置为y + 1，向服务器发送ACK报文段，这个报文段发送完毕以后，客户端和服务器端都进入ESTABLISHED状态，完成TCP三次握手。

为什么要三次握手？

为了防止已失效的连接请求报文段突然又传送到了服务端，因而产生错误。

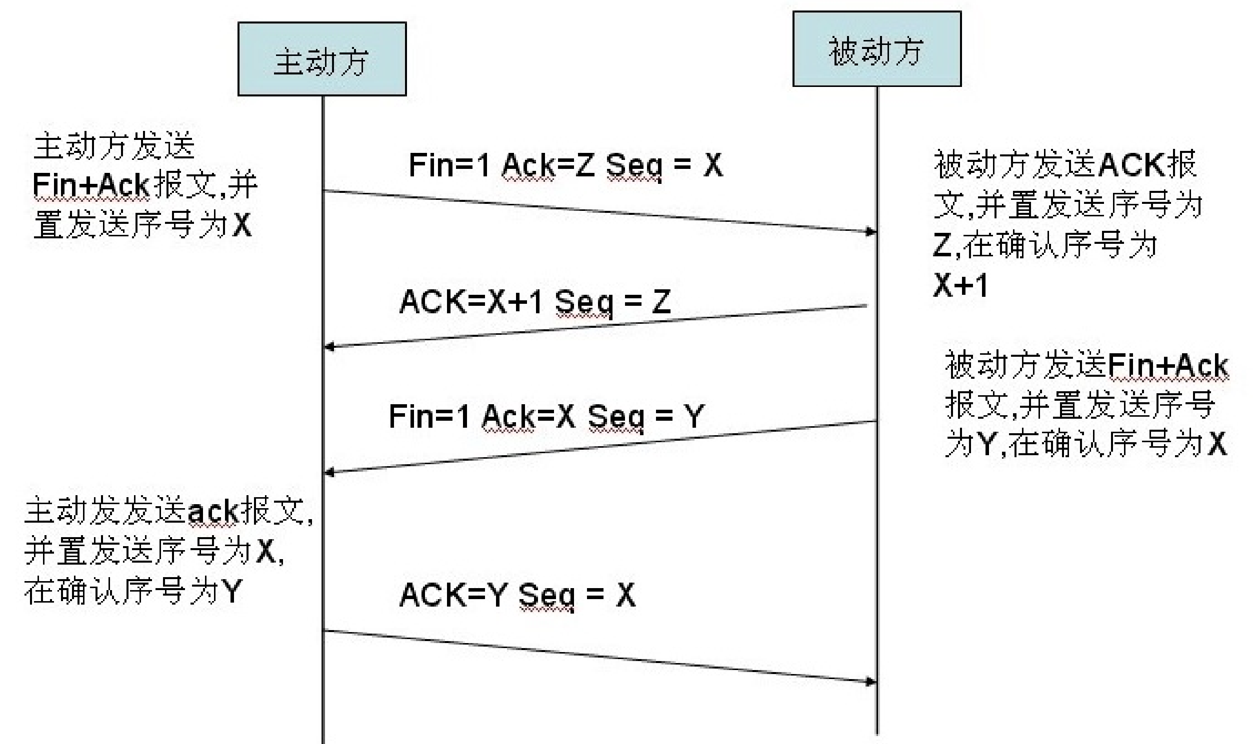
具体例子——“已失效的连接请求报文段”的产生在这样一种情况下：

client发出的第一个连接请求报文段并没有丢失，而是在某个网络结点长时间的滞留了，以致延误到连接释放以后的某个时间才到达server。本来这是一个早已失效的报文段。但server收到此失效的连接请求报文段后，就误认为是client再次发出的一个新的连接请求。于是就向client发出确认报文段，同意建立连接。假设不采用“三次握手”，那么只要server发出确认，新的连接就建立了。由于现在client并没有发出建立连接的请求，因此不会理睬server的确认，也不会向server发送数据。但server却以为新的运输连接已经建立，并一直等待client发来数据。这样，server的很多资源就白白浪费掉了。采用“三次握手”的办法可以防止上述现象发生。

例如刚才那种情况，client不会向server的确认发出确认。server由于收不到确认，就知道client并没有要求建立连接。”

2.四次挥手

当客户端和服务器通过三次握手建立了TCP连接以后，当数据传送完毕，肯定是要断开TCP连接的啊。那对于TCP的断开连接，这里就有了神秘的“四次分手”。



回合	说明
第一次分手	主机1（可以使客户端，也可以是服务器端），设置Sequence Number，向主机2发送一个FIN报文段；此时，主机1进入FIN_WAIT_1状态；这表示主机1没有数据要发送给主机2了；
第二次分手	主机2收到了主机1发送的FIN报文段，向主机1回一个ACK报文段，Acknowledgment Number为 Sequence Number加1；主机1进入FIN_WAIT_2状态；主机2告诉主机1，我“同意”你的关闭请求；
第三次分手	主机2向主机1发送FIN报文段，请求关闭连接，同时主机2进入LAST_ACK状态；
第四次分手	主机1收到主机2发送的FIN报文段，向主机2发送ACK报文段，然后主机1进入TIME_WAIT状态；主机2收到主机1的ACK报文段以后，就关闭连接；此时，主机1等待2MSL（MSL：Maximum Segment Lifetime，报文段最大生存时间，详情请见下文介绍）后依然没有收到回复，则证明Server端已正常关闭，那好，主机1也可以关闭连接了。

为什么要四次分手？

TCP协议是一种面向连接的、可靠的、基于字节流的运输层通信协议。TCP是全双工模式，这就意味着，当主机1发出FIN报文段时，只是表示主机1已经没有数据要发送了，主机1告诉主机2，它的数据已经全部发送完毕了；但是，这个时候主机1还是可以接受来自主机2的数据；当主机2返回ACK报文段时，表示它已经知道主机1没有数据发送了，但是主机2还是可以发送数据到主机1的；当主机2也发送了FIN报文段时，这个时候就表示主机2也没有数据要发送了，就会告诉主机1，我也没有数据要发送了，之后彼此就会愉快的中断这次TCP连接。

为什么要等待**2MSL**？

MSL：报文段最大生存时间，它是任何报文段被丢弃前在网络内的最长时间。
原因有二：

- 保证TCP协议的全双工连接能够可靠关闭
- 保证这次连接的重复数据段从网络中消失

第一点：如果主机1直接CLOSED了，那么由于IP协议的不可靠性或者是其它网络原因，导致主机2没有收到主机1最后回复的ACK。那么主机2就会在超时之后继续发送FIN，此时由于主机1已经CLOSED了，就找不到与重发的FIN对应的连接。所以，主机1不是直接进入CLOSED，而是要保持TIME_WAIT状态。当再次收到FIN的时候，能够保证对方收到ACK，最后正确的关闭连接。

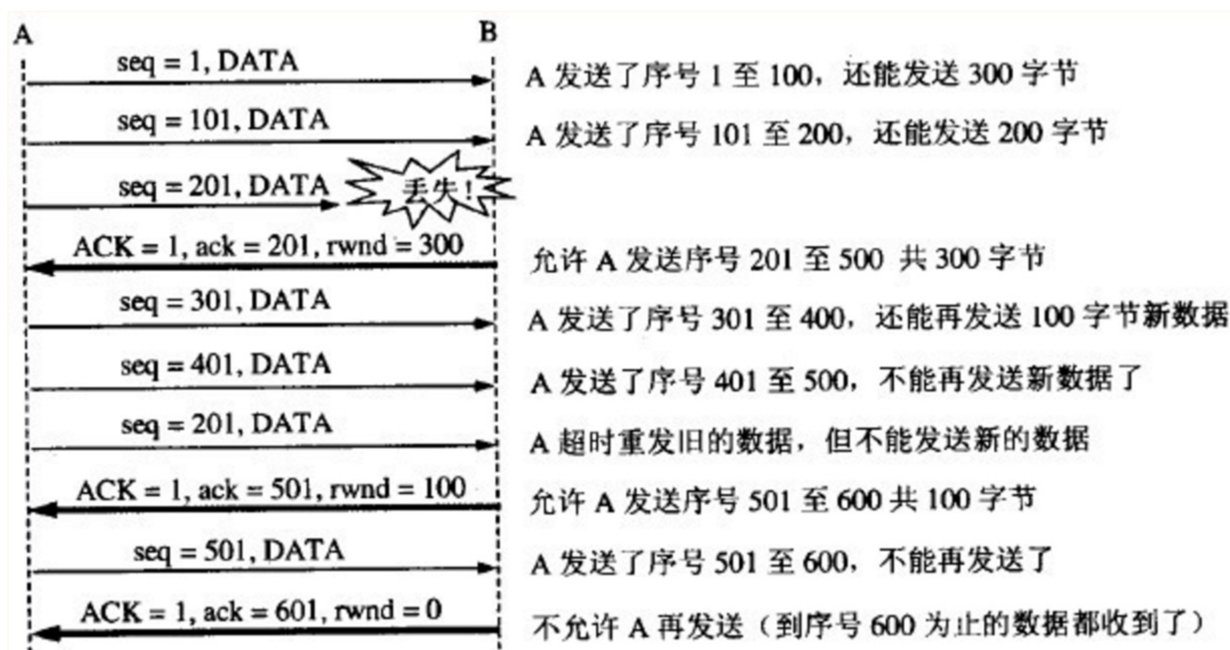
第二点：如果主机1直接CLOSED，然后又再向主机2发起一个新连接，我们不能保证这个新连接与刚关闭的连接的端口号是不同的。也就是说有可能新连接和老连接的端口号是相同的。一般来说不会发生什么问题，但是还是有特殊情况出现：假设新连接和已经关闭的老连接端口号是一样的，如果前一次连接的某些数据仍然滞留在网络中（称为Lost Duplicate），这些延迟数据在建立新连接之后才到达主机2，由于新连接和老连接的端口号是一样的，TCP协议就认为那个延迟的数据是属于新连接的，这样就和真正的新连接的数据包发生混淆了。所以TCP连接要在TIME_WAIT状态等待2倍MSL，保证本次连接的所有数据都从网络中消失。

九、TCP流量控制

如果发送方把数据发送得过快，接收方可能会来不及接收，这就会造成数据的丢失。所谓流量控制就是让发送方的发送速率不要太快，要让接收方来得及接收。

利用滑动窗口机制可以很方便地在TCP连接上实现对发送方的流量控制。

设A向B发送数据。在连接建立时，B告诉了A：“我的接收窗口是 rwnd = 400 ”(这里的 rwnd 表示 receiver window) 。因此，发送方的发送窗口不能超过接收方给出的接收窗口的数值。请注意，TCP的窗口单位是字节，不是报文段。假设每一个报文段为100字节长，而数据报文段序号的初始值设为1。大写**ACK**表示首部中的确认位**ACK**，小写**ack**表示确认字段的值**ack**。



从图中可以看出，B进行了三次流量控制。第一次把窗口减少到 rwnd = 300 ，第二次又减到了 rwnd = 100 ，最后减到 rwnd = 0 ，即不允许发送方再发送数据了。这种使发送方暂停发送的状态将持续到主机B重新发出一个新的窗口值为止。B向A发送的三个报文段都设置了 ACK = 1 ，只有在ACK=1时确认号字段才有意义。

TCP为每一个连接设有一个持续计时器(persistence timer)。当TCP连接中的发送方收到接收方的零窗口通知时，发送方就启动持续计时器。若持续计时器设置的时间到期，发送方就发送一个零窗口控测报文段（携1字节的数据）给接收方。如果接收方可以接收数据，就重新开始发送数据；如果接收方不能接收数据，就重新设置持续计时器。

十、TCP拥塞控制

1.慢开始和拥塞避免

发送方维持一个拥塞窗口 cwnd (congestion window)的状态变量。拥塞窗口的大小取决于网络的拥塞程度，并且动态地在变化。发送方让自己的发送窗口等于拥塞窗口。

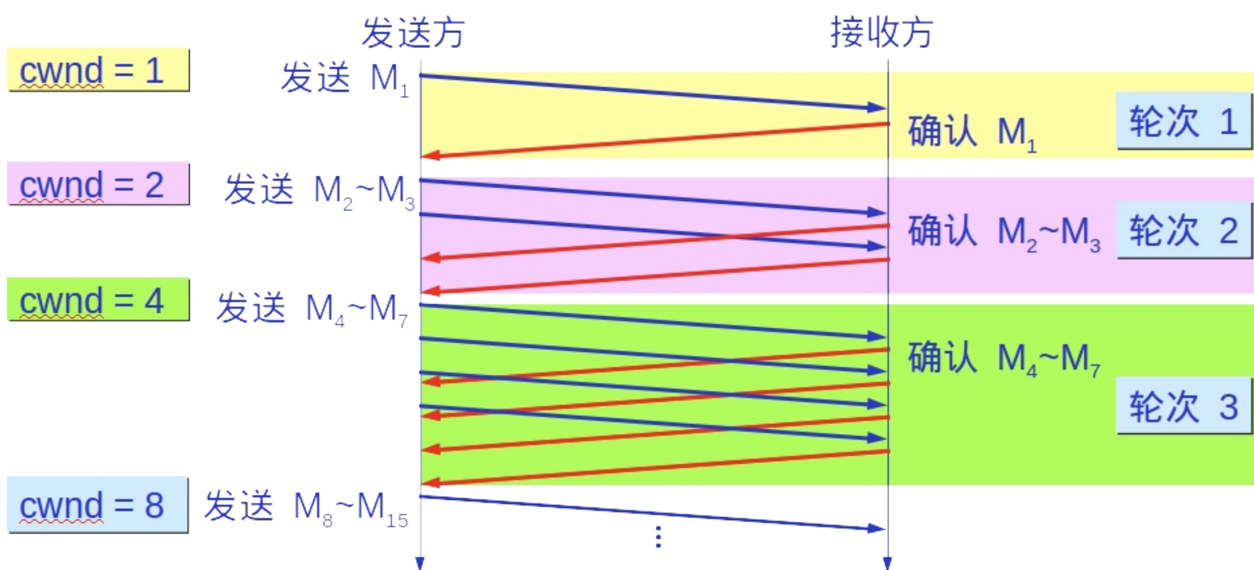
发送方控制拥塞窗口的原则是：只要网络没有出现拥塞，拥塞窗口就再增大一些，以便把更多的分组发送出去。但只要网络出现拥塞，拥塞窗口就减小一些，以减少注入到网络中的分组数。

慢开始算法：

当主机开始发送数据时，如果立即所大量数据字节注入到网络，那么就有可能引起网络拥塞，因为现在并不清楚网络的负荷情况。

因此，较好的方法是，先探测一下，即由小到大逐渐增大发送窗口，也就是说，由小到大逐渐增大拥塞窗口数值。

通常在刚刚开始发送报文段时，先把拥塞窗口 cwnd 设置为一个最大报文段 MSS（Maximum Segment Size，最大报文长度）的数值。而在每收到一个对新的报文段的确认后，把拥塞窗口增加至多一个MSS的数值（底数为2的指数增长规律）。用这样的方法逐步增大发送方的拥塞窗口 cwnd ，可以使分组注入到网络的速率更加合理。



每经过一个传输轮次，拥塞窗口 cwnd 就加倍。

一个传输轮次所经历的时间其实就是往返时间**RTT**。

不过“传输轮次”更加强调：把拥塞窗口cwnd所允许发送的报文段都连续发送出去，并收到了对已发送的最后一个字节的确认。

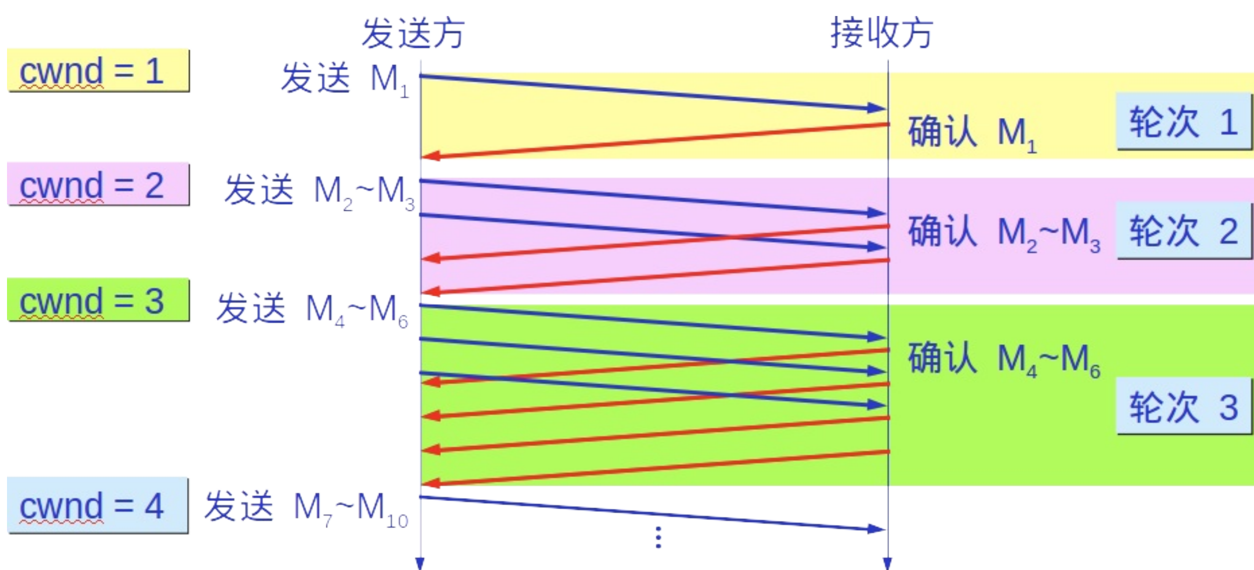
另，慢开始的“慢”并不是指cwnd的增长速率慢，而是指在TCP开始发送报文段时先设置cwnd=1，使得发送方在开始时只发送一个报文段（目的是试探一下网络的拥塞情况），然后再逐渐增大cwnd。

为了防止拥塞窗口cwnd增长过大引起网络拥塞，还需要设置一个慢开始门限ssthresh状态变量。慢开始门限ssthresh的用法如下：

- 当 cwnd < ssthresh 时，使用上述的慢开始算法。
- 当 cwnd > ssthresh 时，停止使用慢开始算法而改用拥塞避免算法。
- 当 cwnd = ssthresh 时，既可使用慢开始算法，也可使用拥塞控制避免算法。

拥塞避免

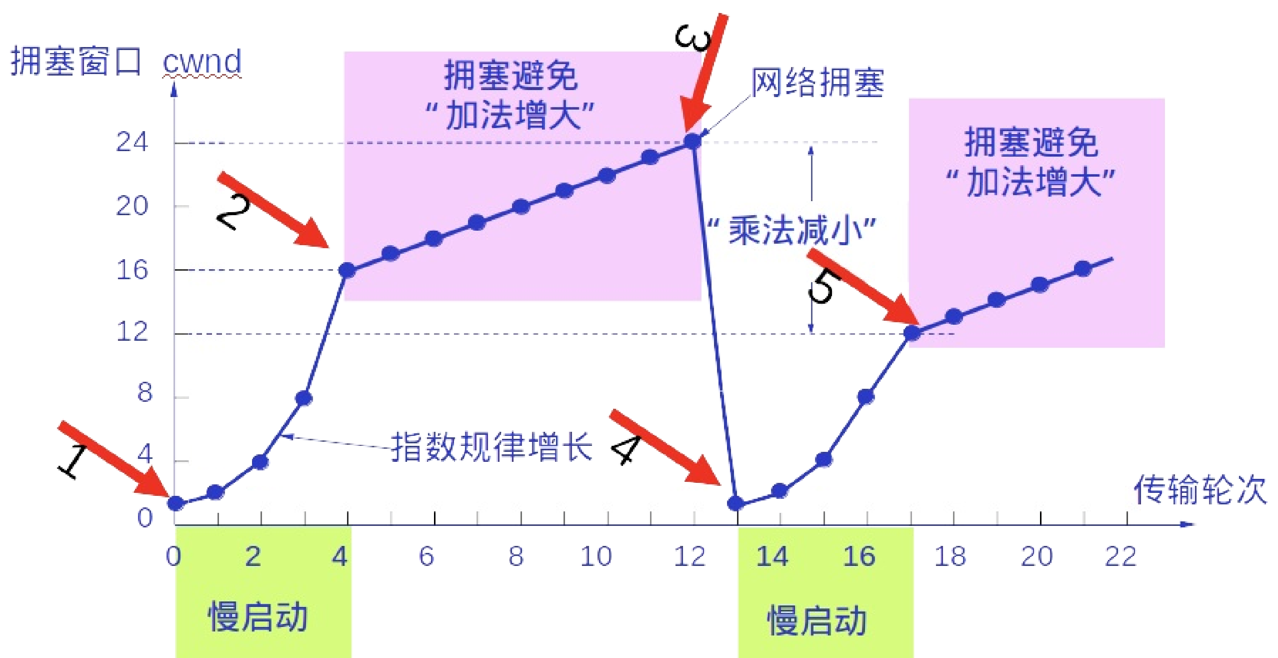
让拥塞窗口cwnd缓慢地增大，即每经过一个往返时间**RTT**就把发送方的拥塞窗口**cwnd**加**1**，而不是加倍。这样拥塞窗口cwnd按线性增长规律缓慢增长，比慢开始算法的拥塞窗口增长速率缓慢得多。



无论在慢开始阶段还是在拥塞避免阶段，只要发送方判断网络出现拥塞（其根据就是没有收到确认），就要把慢开始门限ssthresh设置为出现拥塞时的发送 方窗口值的一半（但不能小于2）。然后把拥塞窗口cwnd重新设置为1，执行慢开始算法。

这样做的目的就是要迅速减少主机发送到网络中的分组数，使得发生 拥塞的路由器有足够时间把队列中积压的分组处理完毕。

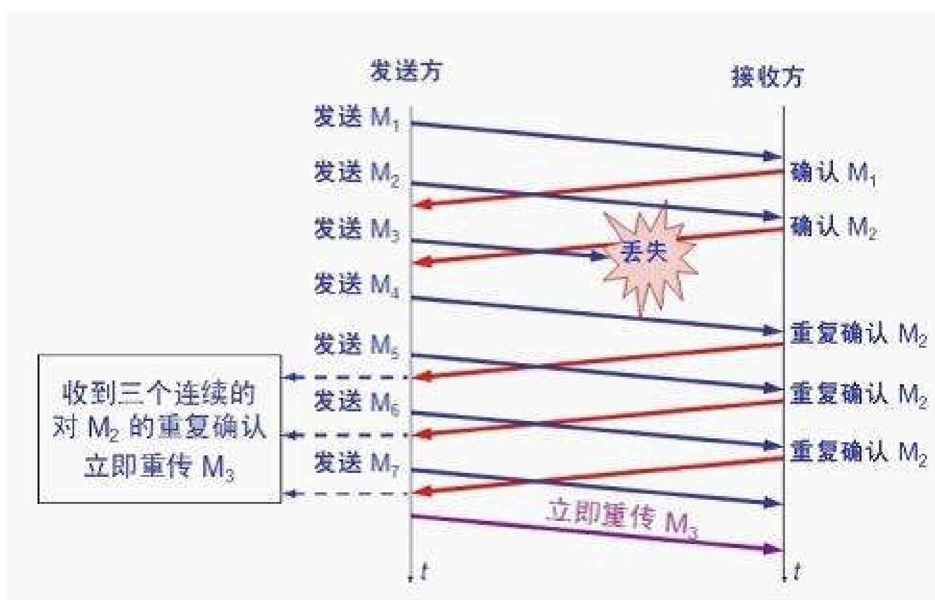
如下图，用具体数值说明了上述拥塞控制的过程。现在发送窗口的大小和拥塞窗口一样大。



2.快重传和快恢复

快重传

快重传算法首先要求接收方每收到一个失序的报文段后就立即发出重复确认（为的是使发送方及早知道有报文段没有到达对方）而不要等到自己发送数据时才进行捎带确认。



接收方收到了M1和M2后都分别发出了确认。现在假定接收方没有收到M3但接着收到了M4。

显然，接收方不能确认M4，因为M4是收到的失序报文段。根据 可靠传输原理，接收方可以什么都不做，也可以在适当时机发送一次对M2的确认。

但按照快重传算法的规定，接收方应及时发送对M2的重复确认，这样做可以让发送方及早知道报文段M3没有到达接收方。发送方接着发送了M5和M6。接收方收到这两个报文后，也还要再次发出对M2的重复确认。这样，发送方共收到了接收方的四个对M2的确认，其中后三个都是重复确认。

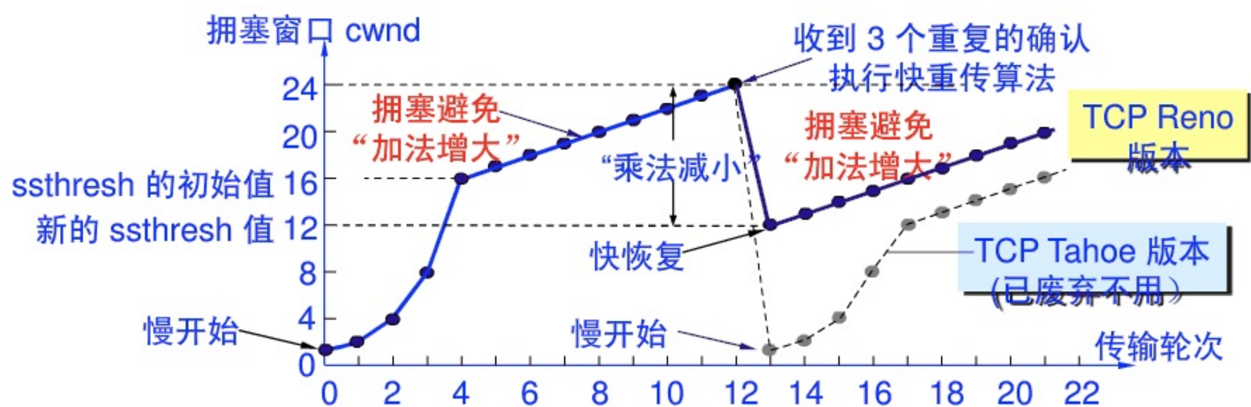
快重传算法还规定，发送方只要一连收到三个重复确认就应当立即重传对方尚未收到的报文段**M3**，而不必 继续等待**M3**设置的重传计时器到期。

由于发送方尽早重传未被确认的报文段，因此采用快重传后可以使整个网络吞吐量提高约20%。

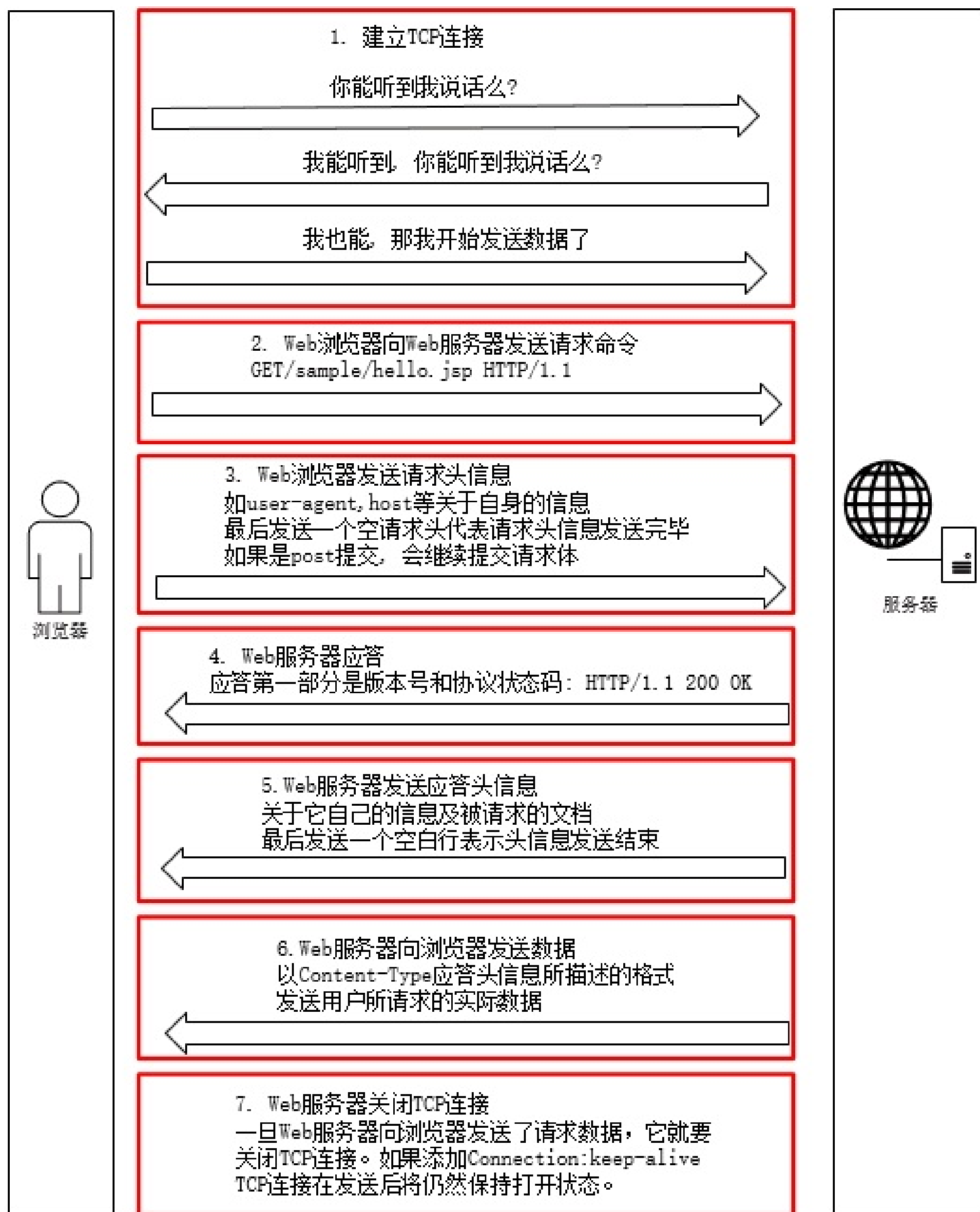
快恢复

与快重传配合使用的还有快恢复算法，其过程有以下两个要点：

- 当发送方连续收到三个重复确认，就执行“乘法减小”算法，把慢开始门限ssthresh减半。
- 与慢开始不同之处是现在不执行慢开始算法（即拥塞窗口cwnd现在不设置为1），而是把cwnd值设置为 慢开始门限ssthresh减半后的数值，然后开始执行拥塞避免算法（“加法增大”），使拥塞窗口缓慢地线性增大。



一、HTTP请求和响应步骤



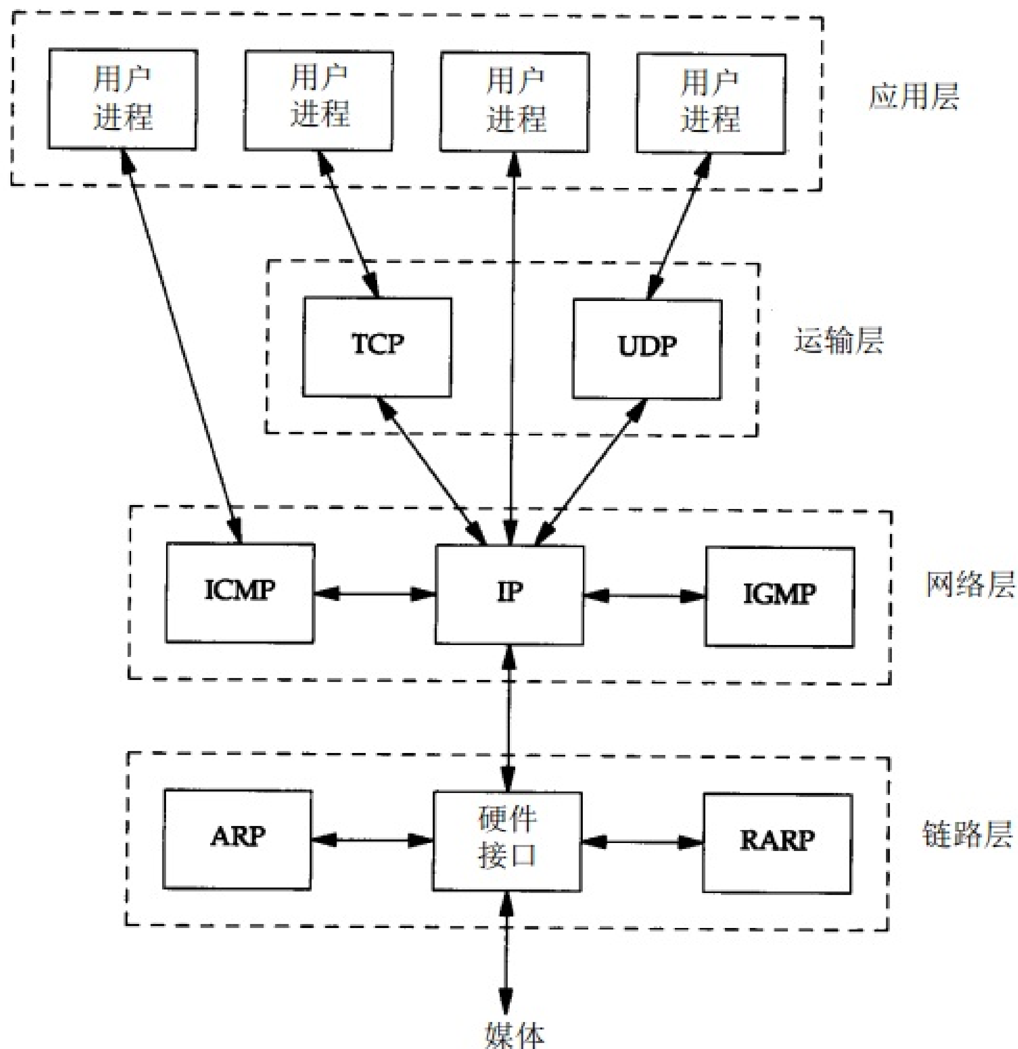
以上完整表示了HTTP请求和响应的7个步骤，下面从TCP/IP协议模型的角度来理解HTTP请求和响应如何传递的。

二、TCP/IP协议

TCP/IP协议模型（Transmission Control Protocol/Internet Protocol），包含了一系列构成互联网基础的网络协议，是Internet的核心协议，通过20多年的发展已日渐成熟，并被广泛应用于局域网和广域网中，目前已成为事实上的国际标准。TCP/IP

协议簇是一组不同层次上的多个协议的组合，通常被认为是一个四层协议系统，与OSI的七层模型相对应。

HTTP协议就是基于TCP/IP协议模型来传输信息的。



TCP/IP协议族中不同层次的协议

(1). 链路层

也称作数据链路层或网络接口层（在第一个图中为网络接口层和硬件层），通常包括操作系统中的设备驱动程序和计算机中对应的网络接口卡。它们一起处理与电缆（或其他任何传输媒介）的物理接口细节。ARP（地址解析协议）和RARP（逆地址解析协议）是某些网络接口（如以太网和令牌环网）使用的特殊协议，用来转换IP层和网络接口层使用的地址。

(2). 网络层

也称作互联网层（在第一个图中为网际层），处理分组在网络中的活动，例如分组的选路。在TCP/IP协议族中，网络层协议包括IP协议（网际协议），ICMP协议（Internet互联网控制报文协议），以及IGMP协议（Internet组管理协议）。

IP是一种网络层协议，提供的是一种不可靠的服务，它只是尽可能快地把分组从源结点送到目的结点，但是并不提供任何可靠性保证。同时被TCP和UDP使用。TCP和UDP的每组数据都通过端系统和每个中间路由器中的IP层在互联网中进行传输。

ICMP是IP协议的附属协议。IP层用它来与其他主机或路由器交换错误报文和其他重要信息。

IGMP是Internet组管理协议。它用来把一个UDP数据报多播到多个主机。

(3). 传输层

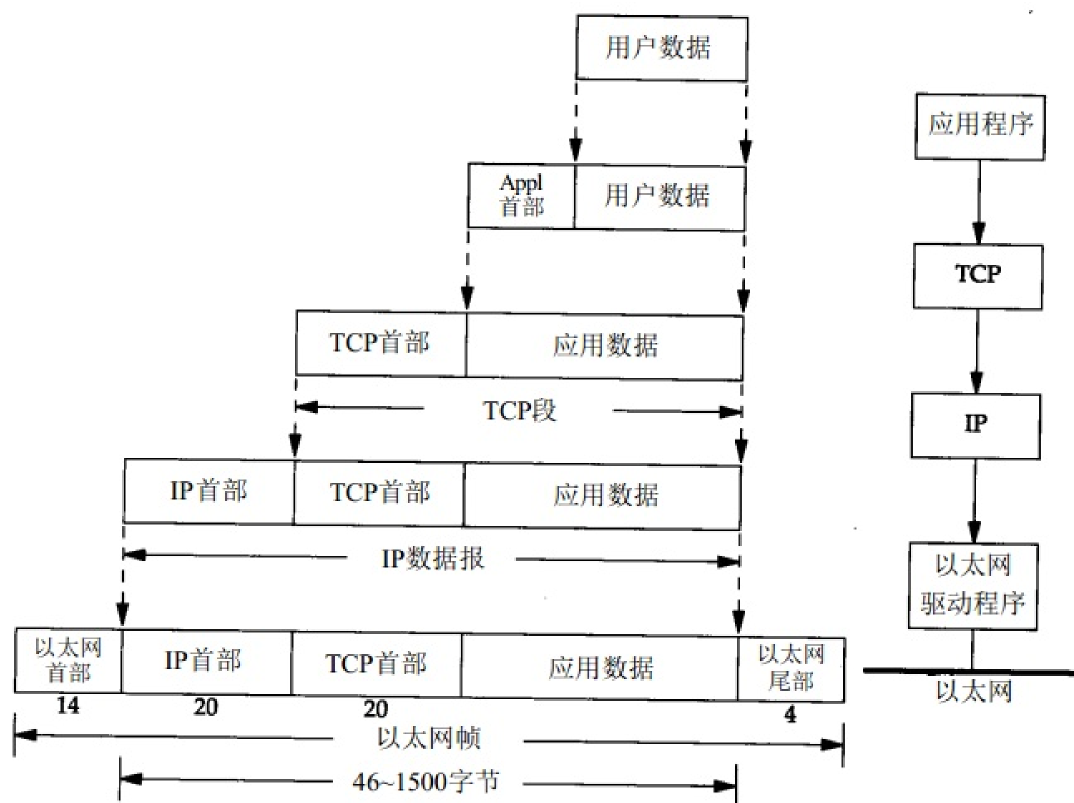
主要为两台主机上的应用程序提供端到端的通信。在TCP/IP协议族中，有两个互不相同的传输协议：TCP（传输控制协议）和UDP（用户数据报协议）。

TCP为两台主机提供高可靠性的数据通信。它所做的工作包括把应用程序交给它的数据分成合适的小块交给下面的网络层，确认接收到的分组，设置发送最后确认分组的超时时钟等。由于运输层提供了高可靠性的端到端的通信，因此应用层可以忽略所有这些细节。为了提供可靠的服务，TCP采用了超时重传、发送和接收端到端的确认分组等机制。

UDP则为应用层提供一种非常简单的服务。它只是把称作数据报的分组从一台主机发送到另一台主机，但并不保证该数据报能到达另一端。一个数据报是指从发送方传输到接收方的一个信息单元（例如，发送方指定的一定字节数的信息）。UDP协议任何必需的可靠性必须由应用层来提供。

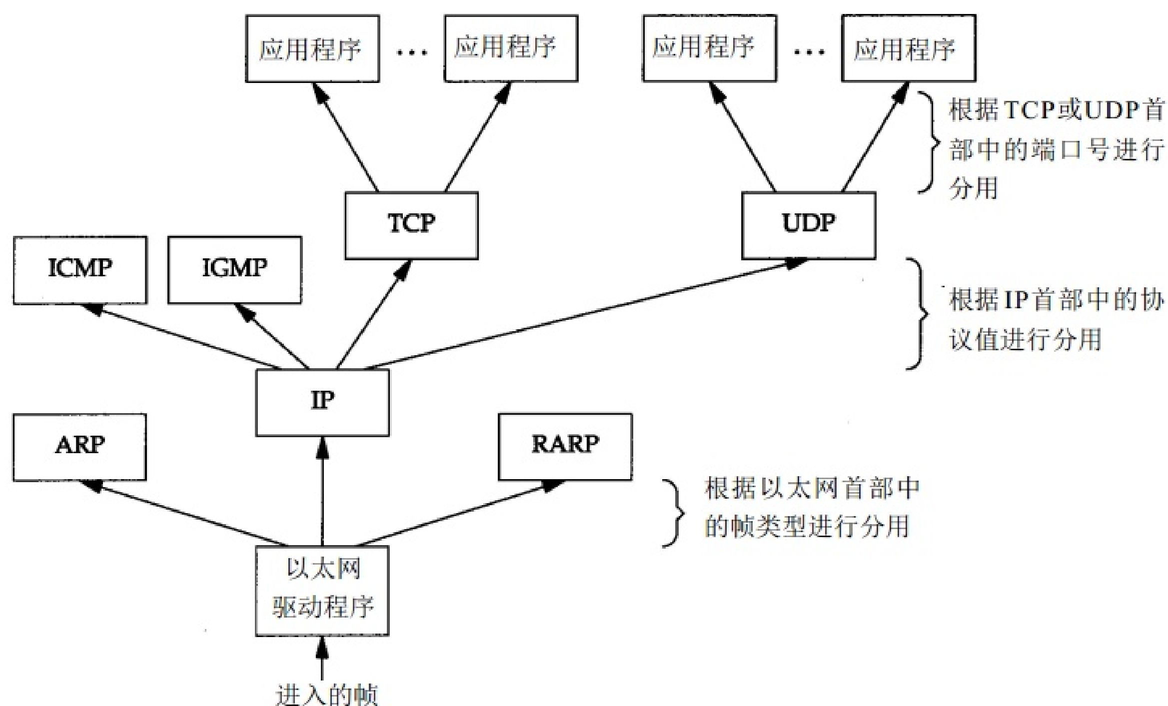
(4). 应用层

应用层决定了向用户提供应用服务时通信的活动。TCP/IP 协议族内预存了各类通用的应用服务。包括 HTTP，FTP（File Transfer Protocol，文件传输协议），DNS（Domain Name System，域名系统）服务。



数据进入协议栈时的封装过程

当应用程序用TCP传送数据时，数据被送入协议栈中，然后逐个通过每一层直到被当作一串比特流送入网络。其中每一层对收到的数据都要增加一些首部信息（有时还要增加尾部信息），该过程如图所示。

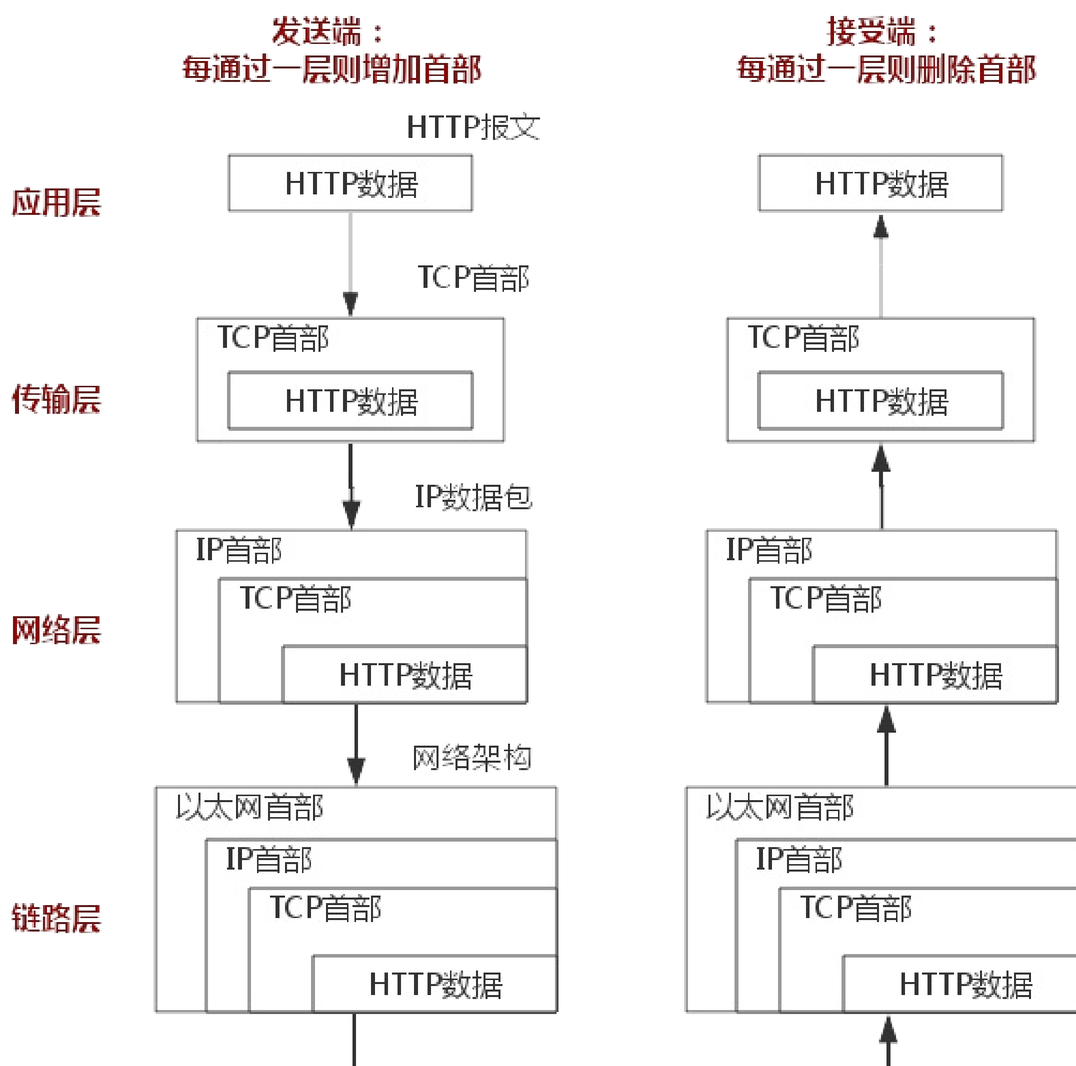


以太网数据帧的分用过程

当目的主机收到一个以太网数据帧时，数据就开始从协议栈中由底向上升，同时去掉各层协议加上的报文首部。每层协议盒都要去检查报文首部中的协议标识，以确定接收数据的上层协议。这个过程称作分用（Demultiplexing）。协议是通过目的端口号、源I P地址和源端口号进行解包的。

通过以上步骤我们从TCP/IP模型的角度来理解了一次HTTP请求与响应的过程。

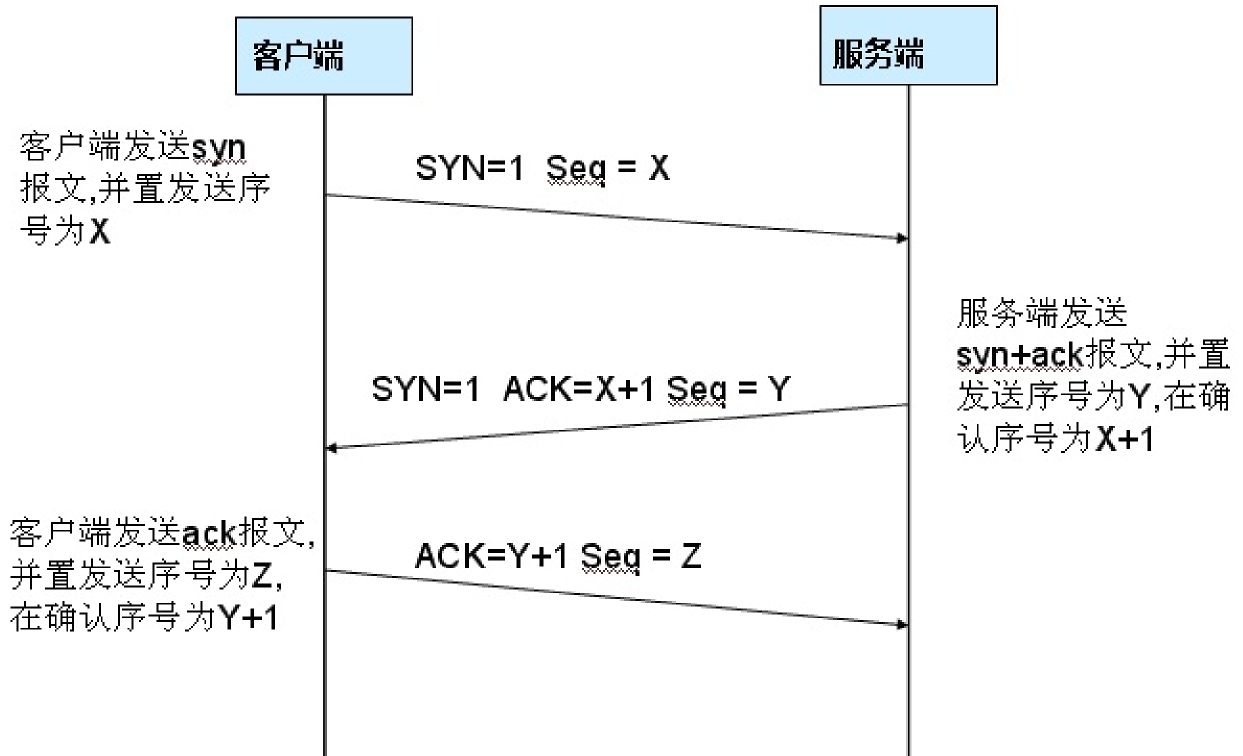
下面这张图更清楚明白：



下面具体来看如何进行一步步操作的。

三、TCP三次握手

TCP是面向连接的，无论哪一方向另一方发送数据之前，都必须先在双方之间建立一条连接。在TCP/IP协议中，TCP协议提供可靠的连接服务，连接是通过三次握手进行初始化的。三次握手的目的是同步连接双方的序列号和确认号并交换 TCP窗口大小信息。



回合	说明
第一次握手	建立连接。客户端发送连接请求报文段，将SYN位置为1，Sequence Number为x；然后，客户端进入SYN_SEND状态，等待服务器的确认；
第二次握手	服务器收到SYN报文段。服务器收到客户端的SYN报文段，需要对这个SYN报文段进行确认，设置Acknowledgment Number为x + 1(Sequence Number + 1)；同时，自己自己还要发送SYN请求信息，将SYN位置为1，Sequence Number为y；服务器端将上述所有信息放到一个报文段（即SYN + ACK报文段）中，一并发送给客户端，此时服务器进入SYN_RECV状态；
第三次握手	客户端收到服务器的SYN + ACK报文段。然后将Acknowledgment Number设置为y + 1，向服务器发送ACK报文段，这个报文段发送完毕以后，客户端和服务器端都进入ESTABLISHED状态，完成TCP三次握手。

为什么要三次握手

为了防止已失效的连接请求报文段突然又传送到了服务端，因而产生错误。

具体例子：——“已失效的连接请求报文段”的产生在这样一种情况下：

client发出的第一个连接请求报文段并没有丢失，而是在某个网络结点长时间的滞留了，以致延误到连接释放以后的某个时间才到达server。本来这是一个早已失效的报文段。但server收到此失效的连接请求报文段后，就误认为是client再次发出的一个新的连接请求。于是就向client发出确认报文段，同意建立连接。假设不采用“三次握手”，那么只要server发出确认，新的连接就建立了。由于现在client并没有发出建立连接的请求，因此不会理睬server的确认，也不会向server发送数据。但server却以为新的运输连接已经建立，并一直等待client发来数据。这样，server的很多资源就白白浪费掉了。采用“三次握手”的办法可以防止上述现象发生。

例如刚才那种情况，client不会向server的确认发出确认。server由于收不到确认，就知道client并没有要求建立连接。”

四、HTTP协议

Http是什么？

通俗来讲，他就是计算机通过网络进行通信的规则，是一个基于请求与响应，无状态的，应用层的协议，常基于TCP/IP协议传输数据。目前任何终端（手机，笔记本电脑。。）之间进行任何一种通信都必须按照Http协议进行，否则无法连接。

四个基于：

请求与响应：客户端发送请求，服务器端响应数据

无状态的：协议对于事务处理没有记忆能力，客户端第一次与服务器建立连接发送请求时需要进行一系列的安全认证匹配等，因此增加页面等待时间，当客户端向服务器端发送请求，服务器端响应完毕后，两者断开连接，也不保存连接状态，一刀两断！恩断义绝！从此路人！下一次客户端向同样的服务器发送请求时，由于他们之前已经遗忘了彼此，所以需要重新建立连接。

应用层：Http是属于应用层的协议，配合TCP/IP使用。

TCP/IP：Http使用TCP作为它的支撑运输协议。HTTP客户机发起一个与服务器的TCP连接，一旦连接建立，浏览器（客户机）和服务器进程就可以通过套接字接口访问TCP。

针对无状态的一些解决策略：

有时需要对用户之前的HTTP通信状态进行保存，比如执行一次登陆操作，在30分钟内所有的请求都不需要再次登陆。于是引入了Cookie技术。

HTTP/1.1想出了持久连接（HTTP keep-alive）方法。其特点是，只要任意一端没有明确提出断开连接，则保持TCP连接状态，在请求首部字段中的Connection: keep-alive即为表明使用了持久连接。等等还有很多……

下面开始讲解重头戏：HTTP请求报文，响应报文，对应于上述步骤的2，3，4，5，6。

HTTP报文是面向文本的，报文中的每一个字段都是一些ASCII码串，各个字段的长度是不确定的。HTTP有两类报文：请求报文和响应报文。

五、HTTP请求报文

一个HTTP请求报文由请求行（request line）、请求头部（header）、空行和请求数据4个部分组成，下图给出了请求报文的一般格式。



1.请求行

请求行分为三个部分：请求方法、请求地址和协议版本

请求方法

HTTP/1.1 定义的请求方法有8种：GET、POST、PUT、DELETE、PATCH、HEAD、OPTIONS、TRACE。

最常的两种GET和POST，如果是RESTful接口的话一般会用到GET、POST、DELETE、PUT。

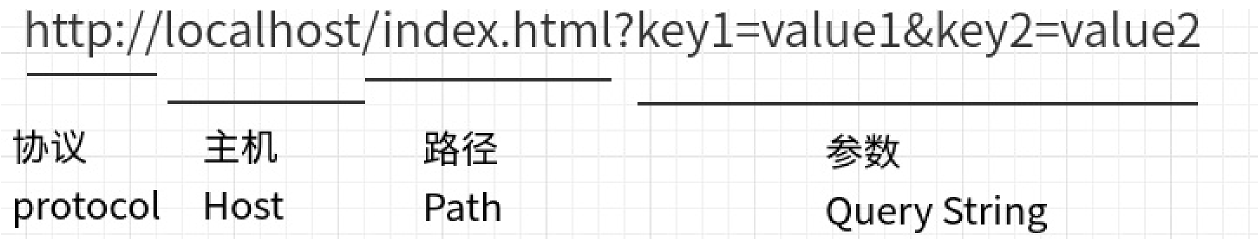
请求地址

URL：统一资源定位符，是一种自愿位置的抽象唯一识别方法。

组成：<协议>：//<主机>：<端口>/<路径>

端口和路径有时可以省略（**HTTP**默认端口号是**80**）

如下例：



有时会带参数，GET请求

协议版本

协议版本的格式为：HTTP/主版本号.次版本号，常用的有HTTP/1.0和HTTP/1.1

2.请求头部

请求头部为请求报文添加了一些附加信息，由“名/值”对组成，每行一对，名和值之间使用冒号分隔。

常见请求头如下：

请求头	说明
Host	接受请求的服务器地址，可以是IP:端口号，也可以是域名
User-Agent	发送请求的应用程序名称
Connection	指定与连接相关的属性，如Connection:Keep-Alive
Accept-Charset	通知服务端可以发送的编码格式
Accept-Encoding	通知服务端可以发送的数据压缩格式
Accept-Language	通知服务端可以发送的语言

请求头部的最后会有一个空行，表示请求头部结束，接下来为请求数据，这一行非常重要，必不可少。

3.请求数据

可选部分，比如GET请求就没有请求数据。

下面是一个POST方法的请求报文：

> POST /index.php HTTP/1.1 请求行
> Host: localhost
> User-Agent: Mozilla/5.0 (Windows NT 5.1; rv:10.0.2) Gecko/20100101 Firefox/10.0.2 请求头
> Accept: text/html,application/xhtml+xml,application/xml;q=0.9,/;q=0.8
> Accept-Language: zh-cn,zh;q=0.5
> Accept-Encoding: gzip, deflate
> Connection: keep-alive
> Referer:http://localhost/
> Content-Length：25
> Content-Type：application/x-www-form-urlencoded
> 空行
> username=aa&password=1234 请求数据

六、HTTP响应报文

协议版本 | 空格 | 状态码 | 空格 | 状态码描述 | 回车符 | 换行符 状态行
头部字段名 | : | 值 | 回车符 | 换行符
...
头部字段名 | : | 值 | 回车符 | 换行符 响应头部
回车符 | 换行符
响应正文

HTTP响应报文主要由状态行、响应头部、空行以及响应数据组成。

1.状态行

由3部分组成，分别为：协议版本，状态码，状态码描述。

其中协议版本与请求报文一致，状态码描述是对状态码的简单描述，所以这里就只介绍状态码。

状态码

状态代码为3位数字。

- 1xx：指示信息——表示请求已接收，继续处理。
- 2xx：成功——表示请求已被成功接收、理解、接受。
- 3xx：重定向——要完成请求必须进行更进一步的操作。

- 4xx：客户端错误——请求有语法错误或请求无法实现。
- 5xx：服务器端错误——服务器未能实现合法的请求。

下面列举几个常见的：

状态码	说明
200	响应成功
302	跳转，跳转地址通过响应头中的Location属性指定（JSP中Forward和Redirect之间的区别）
400	客户端请求有语法错误，不能被服务器识别
403	服务器接收到请求，但是拒绝提供服务（认证失败）
404	请求资源不存在
500	服务器内部错误

2.响应头部

与请求头部类似，为响应报文添加了一些附加信息

常见响应头部如下：

响应头	说明
Server	服务器应用程序软件的名称和版本
Content-Type	响应正文的类型（是图片还是二进制字符串）
Content-Length	响应正文长度
Content-Charset	响应正文使用的编码
Content-Encoding	响应正文使用的数据压缩格式
Content-Language	响应正文使用的语言

3.响应数据

用于存放需要返回给客户端的数据信息。

下面是一个响应报文的实例：

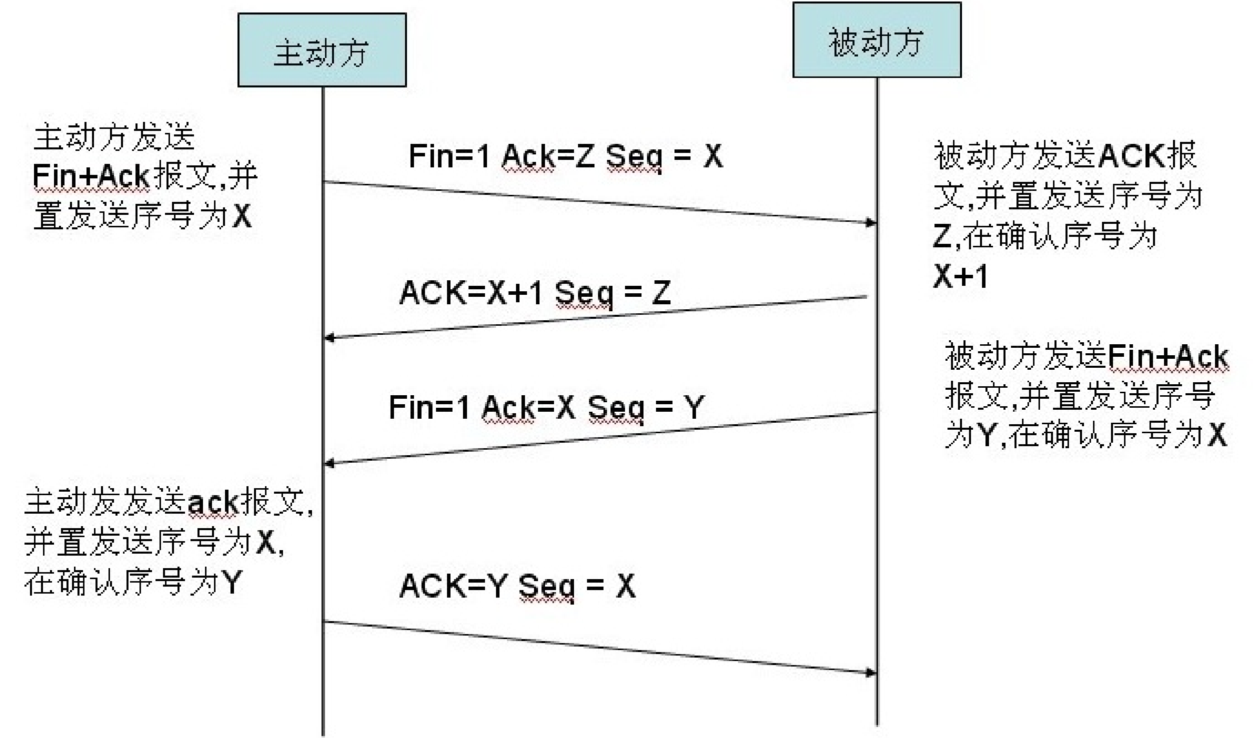
> HTTP/1.1 200 OK　　状态行
> Date: Sun, 17 Mar 2013 08:12:54 GMT　　响应头部
> Server: Apache/2.2.8 (Win32) PHP/5.2.5
> X-Powered-By: PHP/5.2.5
> Set-Cookie: PHPSESSID=c0huq7pdkmm5gg6osoe3mgjmm3; path=/
> Expires: Thu, 19 Nov 1981 08:52:00 GMT
> Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
> Pragma: no-cache
> Content-Length: 4393
> Keep-Alive: timeout=5, max=100
> Connection: Keep-Alive
> Content-Type: text/html; charset=utf-8
> 　　空行
>
> <html>　　响应数据
> <head>
> <title>HTTP响应示例<title>
> </head>
> <body>
> Hello HTTP!
> </body>
> </html>

关于请求头部和响应头部的知识点很多，这里只是简单介绍。

通过以上步骤，数据已经传递完毕，HTTP/1.1会维持持久连接，但持续一段时间总会有关闭连接的时候，这时候据需要断开TCP连接。

七、TCP四次挥手

当客户端和服务器通过三次握手建立了TCP连接以后，当数据传送完毕，肯定是要断开TCP连接的啊。那对于TCP的断开连接，这里就有了神秘的“四次分手”。



回合	说明
第一次分手	主机1（可以使客户端，也可以是服务器端），设置Sequence Number，向主机2发送一个FIN报文段；此时，主机1进入FIN_WAIT_1状态；这表示主机1没有数据要发送给主机2了；
第二次分手	主机2收到了主机1发送的FIN报文段，向主机1回一个ACK报文段，Acknowledgment Number为Sequence Number加1；主机1进入FIN_WAIT_2状态；主机2告诉主机1，我“同意”你的关闭请求；
第三次分手	主机2向主机1发送FIN报文段，请求关闭连接，同时主机2进入LAST_ACK状态；
第四次分手	主机1收到主机2发送的FIN报文段，向主机2发送ACK报文段，然后主机1进入TIME_WAIT状态；主机2收到主机1的ACK报文段以后，就关闭连接；此时，主机1等待2MSL后依然没有收到回复，则证明Server端已正常关闭，那好，主机1也可以关闭连接了。

为什么要四次分手

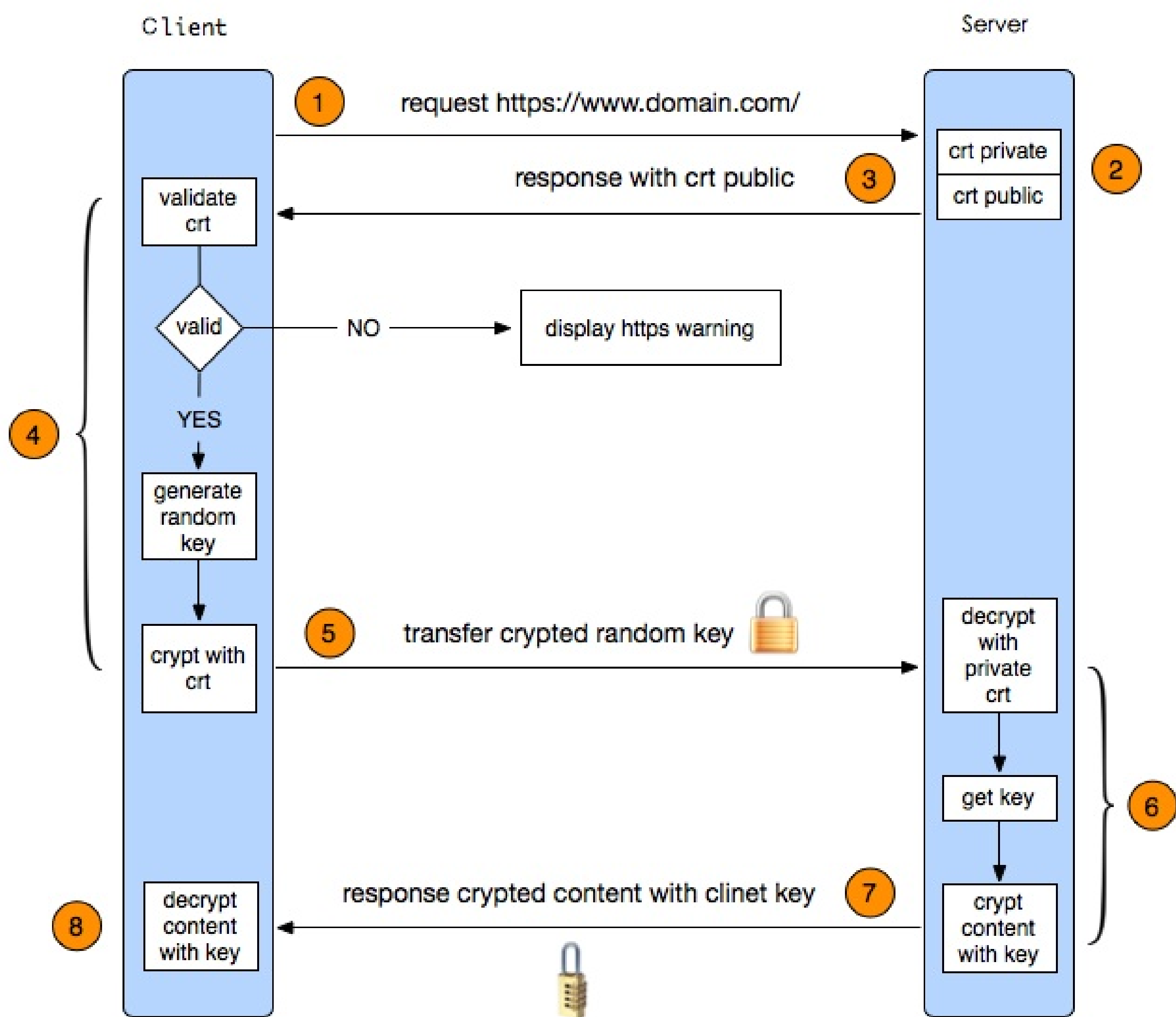
TCP协议是一种面向连接的、可靠的、基于字节流的运输层通信协议。TCP是全双工模式，这就意味着，当主机1发出FIN报文段时，只是表示主机1已经没有数据要发送了，主机1告诉主机2，它的数据已经全部发送完毕了；但是，这个时候主机1还是可以接受来自主机2的数据；当主机2返回ACK报文段时，表示它已经知道主机1没有数据发送了，但是主机2还是可以发送数据到主机1的；当主机2也发送了FIN报文段时，这个时候就表示主机2也没有数据要发送了，就会告诉主机1，我也没有数据要发送了，之后彼此就会愉快的中断这次TCP连接。

通过以上步骤便完成了HTTP的请求和响应，进行了数据传递，这其中涉及到需要知识点，都进行了逐一了解。

我们都知道HTTPS能够加密信息，以免敏感信息被第三方获取。所以很多银行网站或电子邮箱等等安全级别较高的服务都会采用HTTPS协议。

HTTPS简介

HTTPS其实是有两部分组成：HTTP + SSL / TLS，也就是在HTTP上又加了一层处理加密信息的模块。服务端和客户端的信息传输都会通过TLS进行加密，所以传输的数据都是加密后的数据。具体是如何进行加密，解密，验证的，且看下图。



1. 客户端发起HTTPS请求

这个没什么好说的，就是用户在浏览器里输入一个https网址，然后连接到server的443端口。

2. 服务端的配置

采用HTTPS协议的服务器必须要有一套数字证书，可以自己制作，也可以向组织申请。区别就是自己颁发的证书需要客户端验证通过，才可以继续访问，而使用受信任的公司申请的证书则不会弹出提示页面(startssl就是个不错的选择，有1年的免费服务)。这套证书其实就是一对公钥和私钥。如果对公钥和私钥不太理解，可以想象成一把钥匙和一个锁头，只是全世界只有你一个人有这把钥匙，你可以把锁头给别人，别人可以用这个锁把重要的东西锁起来，然后发给你，因为只有你一个人有这把钥匙，所以只有你才能看到被这把锁锁起来的东西。

3. 传送证书

这个证书其实就是公钥，只是包含了很多信息，如证书的颁发机构，过期时间等等。

4. 客户端解析证书

这部分工作是有客户端的TLS来完成的，首先会验证公钥是否有效，比如颁发机构，过期时间等等，如果发现异常，则会弹出一个警告框，提示证书存在问题。如果证书没有问题，那么就生成一个随即值。然后用证书对该随机值进行加密。就好像上面说的，把随机值用锁头锁起来，这样除非有钥匙，不然看不到被锁住的内容。

5. 传送加密信息

这部分传送的是用证书加密后的随机值，目的就是让服务端得到这个随机值，以后客户端和服务端的通信就可以通过这个随机值来进行加密解密了。

6. 服务段加密信息

服务端用私钥解密后，得到了客户端传过来的随机值(私钥)，然后把内容通过该值进行对称加密。所谓对称加密就是，将信息和私钥通过某种算法混合在一起，这样除非知道私钥，不然无法获取内容，而正好客户端和服务端都知道这个私钥，所以只要加密算法够彪悍，私钥够复杂，数据就够安全。

7. 传输加密后的信息

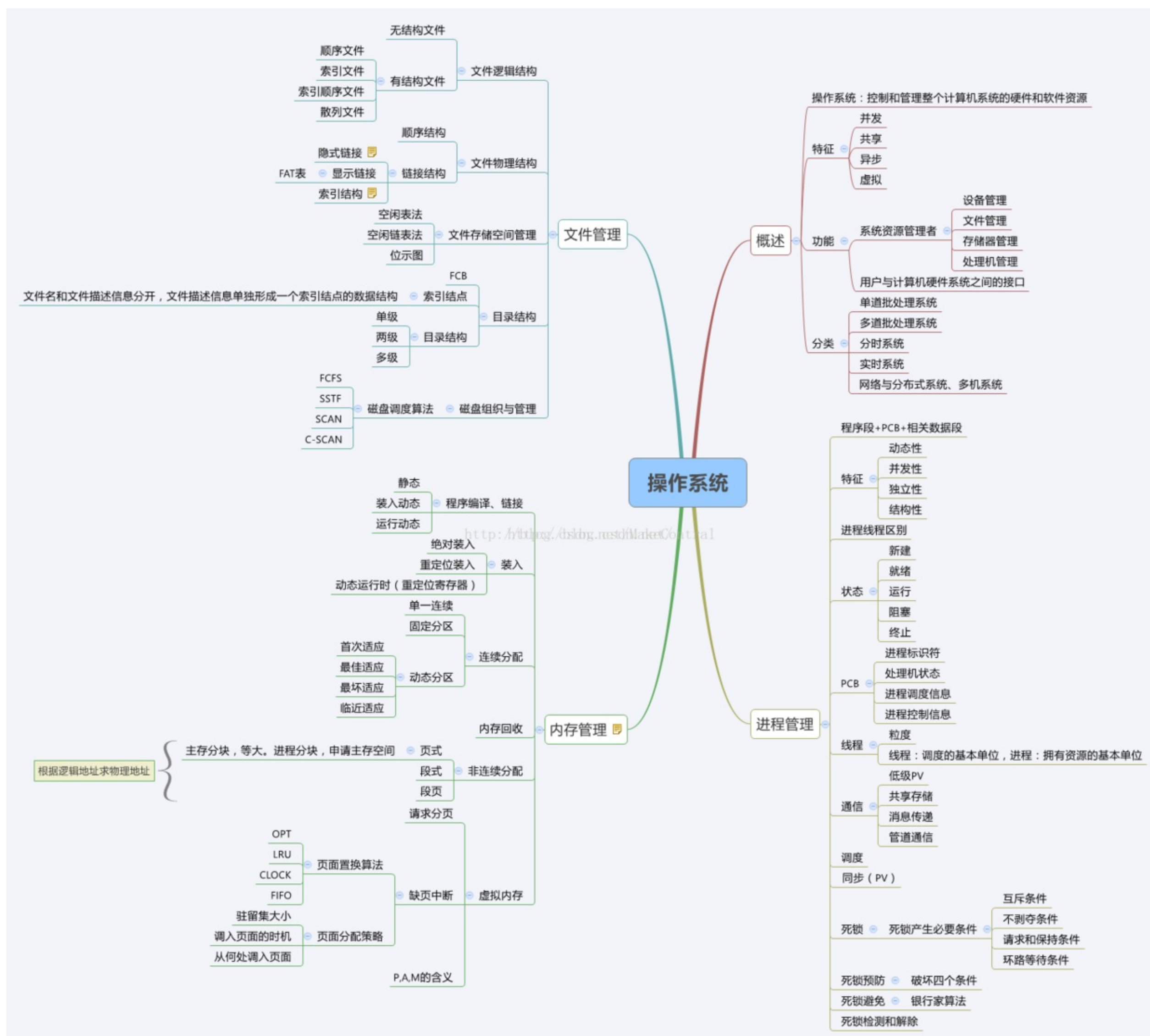
这部分信息是服务段用私钥加密后的信息，可以在客户端被还原

8. 客户端解密信息

客户端用之前生成的私钥解密服务段传过来的信息，于是获取了解密后的内容。整个过程第三方即使监听到了数据，也束手无策。

> 操作系统这一块内容比较晦涩枯燥，如果系统学起来需要耗费很长时间。针对于面试，本章内容整理了一些面试重难点问题，大致分为3块内容，18个问题。

一、操作系统知识点图谱



二、面试问题总结

1. 操作系统的四个特性。
2. 操作系统的主要功能。
3. 进程的有哪几种状态，状态转换图，及导致转换的事件。
4. 进程与线程的区别。
5. 进程通信的几种方式。

6. 进程同步的几种方式
7. 用户态和核心态的区别。
8. 死锁的概念，导致死锁的原因.
9. 导致死锁的四个必要条件。
10. 处理死锁的四个方式。
11. 预防死锁的方法、避免死锁的方法。
12. 进程调度算法。
13. 内存连续分配方式采用的几种算法及各自优劣。
14. 基本分页储存管理方式。
15. 基本分段储存管理方式。
16. 分段分页方式的比较各自优缺点。
17. 几种页面置换算法，会算所需换页数
18. 虚拟内存的定义及实现方式。

三、 概述

1.操作系统的四个特性

并发：同一段时间内多个程序执行(注意区别并行和并发，前者是同一时刻的多个事件，后者是同一时间段内的多个事件)

共享：系统中的资源可以被内存中多个并发执行的进线程共同使用

虚拟：通过时分复用（如分时系统）以及空分复用（如虚拟内存）技术实现把一个物理实体虚拟为多个

异步：系统中的进程是以走走停停的方式执行的，且以一种不可预知的速度推进

2.操作系统的主要功能

处理机管理：处理机分配都是以进程为单位，所以处理机管理也被看做是进程管理。包括进程控制，进程同步，进程通信和进程调度

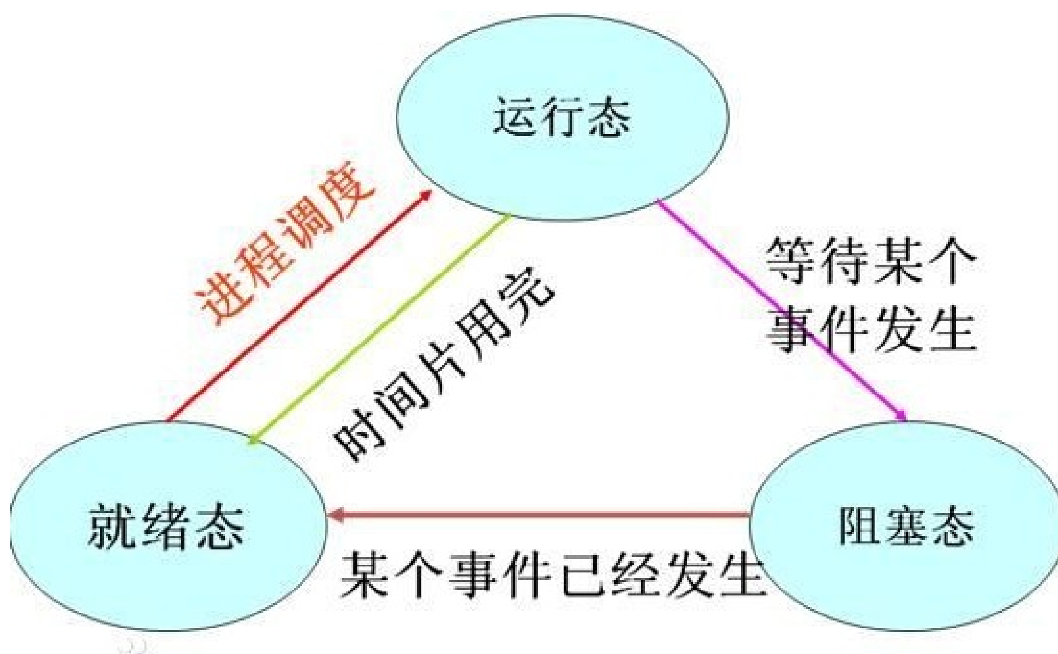
存储器管理（或者内存管理）：内存分配，内存保护，地址映射，内存扩充

设备管理：管理所有外围设备，包括完成用户的IO请求；为用户进程分配IO设备；提高IO设备利用率；提高IO速度；方便IO的使用

文件管理： 管理用户文件和系统文件，方便使用同时保证安全性。包括：磁盘存储空间管理，目录管理，文件读写管理以及文件共享和保护

提供用户接口： 程序接口（如API）和用户接口（如GUI）

一、进程的状态与转换



运行状态：进程正在处理机上运行。在单处理机环境下，每一时刻最多只有一个进程处于运行状态。

就绪状态：进程已处于准备运行的状态，即进程获得了除处理机之外的一切所需资源，一旦得到处理机即可运行。

阻塞状态，又称等待状态：进程正在等待某一事件而暂停运行，如等待某资源为可用（不包括处理机）或等待输入/输出完成。即使处理机空闲，该进程也不能运行。

注意区别就绪状态和等待状态： 就绪状态是指进程仅缺少处理机，只要获得处理机资源就立即执行；而等待状态是指进程需要其他资源（除了处理机）或等待某一事件。

就绪状态 **->** 运行状态： 处于就绪状态的进程被调度后，获得处理机资源（分派处理机时间片），于是进程由就绪状态转换为运行状态。

运行状态 **->** 就绪状态： 处于运行状态的进程在时间片用完后，不得不让出处理机，从而进程由运行状态转换为就绪状态。此外，在可剥夺的操作系统中，当有更高优先级的进程就 、 绪时，调度程度将正执行的进程转换为就绪状态，让更高优先级的进程执行。

运行状态 **->** 阻塞状态： 当进程请求某一资源（如外设）的使用和分配或等待某一事件的发生（如I/O操作的完成）时，它就从运行状态转换为阻塞状态。进程以系统调用的形式请求操作系统提供服务，这是一种特殊的、由运行用户态程序调用操作系统内核过程的形式。

阻塞状态 **->** 就绪状态： 当进程等待的事件到来时，如I/O操作结束或中断结束时，中断处理程序必须把相应进程的状态由阻塞状态转换为就绪状态。

二、进程与线程的区别

进程： 进程是进程实体的运行过程，是系统进行资源分配和调度的一个独立单位（具有动态、并发、独立、异步的特性，以及就绪、执行、阻塞3种状态）；引入进程是为了使多个程序可以并发的执行，以提高系统的资源利用率和吞吐量。

线程： 是比进程更小的可独立运行的基本单位，可以看做是轻量级的进程（具有轻型实体，独立调度分派单位，可并发执行，共享进程资源等属性）；引入目的是为了减少程序在并发执行过程中的开销，使OS的并发效率更高。

两者的对比：

1. 调度方面：在引入线程的OS中，线程是独立的调度和分派单位，而进程作为资源的拥有单位(相当于把未引入线程的传统OS中的进程的两个属性分开了)。由于线程不拥有资源，因此可以显著的提高并发度以及减少切换开销。
2. 并发性：引入了线程的OS中，进程间可以并发，而且一个进程内部的多个线程之间也是可以并发的，这就使OS具有更好的并发性，有效的提高了系统资源利用率和吞吐量。
3. 拥有资源：无论OS是否支持线程，进程都是基本的资源拥有单位，线程只拥有很少的基本的资源，但是线程可以访问所隶属的进程的资源（进程的代码段，数据段和所拥有的系统资源如fd）
4. 系统开销：创建或者撤销进程的时候，系统要为之创建或回收PCB，系统资源等，切换时也需要保存和恢复CPU环境。而线程的切换只需要保存和恢复少量的寄存器，不涉及存储器管理方面的工作，所以开销较小。此外，统一进程中的多个线程由于共享地址空间，所以通信同步等都比较方便。

三、进程通信

进程通信是指进程之间的信息交换。PV操作是低级通信方式，高级通信方式是指以较高的效率传输大量数据的通信方式。高级通信方法主要有以下三个类。

共享存储

在通信的进程之间存在一块可直接访问的共享空间，通过对这片共享空间进行写/读操作实现进程之间的信息交换。在对共享空间进行写/读操作时，需要使用同步互斥工具（如 P操作、V操作），对共享空间的写/读进行控制。共享存储又分为两种：低级方式的共享是基于数据结构的共享；高级方式则是基于存储区的共享。操作系统只负责为通信进程提供可共享使用的存储空间和同步互斥工具，而数据交换则由用户自己安排读/写指令完成。

需要注意的是，用户进程空间一般都是独立的，要想让两个用户进程共享空间必须通过特殊的系统调用实现，而进程内的线程是自然共享进程空间的。

消息传递

在消息传递系统中，进程间的数据交换是以格式化的消息(Message)为单位的。若通信的进程之间不存在可直接访问的共享空间，则必须利用操作系统提供的消息传递方法实现进程通信。进程通过系统提供的发送消息和接收消息两个原语进行数据交换。

1) 直接通信方式：发送进程直接把消息发送给接收进程，并将它挂在接收进程的消息缓冲队列上，接收进程从消息缓冲队列中取得消息。

2) 间接通信方式：发送进程把消息发送到某个中间实体中，接收进程从中间实体中取得消息。这种中间实体一般称为信箱，这种通信方式又称为信箱通信方式。该通信方式广泛应用于计算机网络中，相应的通信系统称为电子邮件系统。

管道通信

管道通信是消息传递的一种特殊方式。所谓“管道”，是指用于连接一个读进程和一个写进程以实现它们之间通信的一个共享文件，又名pipe文件。向管道（共享文件）提供输入的发送进程（即写进程），以字符流形式将大量的数据送入（写）管道；而接收管道输出的接收进程（即读进程），则从管道中接收（读）数据。为了协调双方的通信，管道机制必须提供以下三方面的协调能力：互斥、同步和确定对方的存在。

四、进程同步

多进程虽然提高了系统资源利用率和吞吐量，但是由于进程的异步性可能造成系统的混乱。进程同步的任务就是对多个相关进程在执行顺序上进行协调，使并发执行的多个进程之间可以有效的共享资源和相互合作，保证程序执行的可再现性

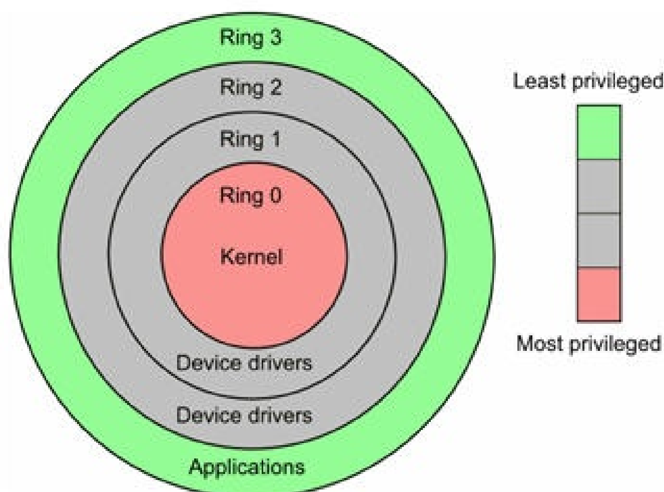
同步机制需要遵循的原则：

1. 空闲让进：当没有进程处于临界区的时候，应该许可其他进程进入临界区的申请
2. 忙则等待：当前如果有进程处于临界区，如果有其他进程申请进入，则必须等待，保证对临界区的互斥访问
3. 有限等待：对要求访问临界资源的进程，需要在有限时间内进入临界区，防止出现死等
4. 让权等待：当进程无法进入临界区的时候，需要释放处理机，边陷入忙等

经典的进程同步问题：生产者-消费者问题；哲学家进餐问题；读者-写者问题

同步的解决方案：管程，信号量。

五、用户态和核心态



当程序运行在**3**级特权级上时，就可以称之为运行在用户态，因为这是最低特权级，是普通的用户进程运行的特权级，大部分用户直接面对的程序都是运行在用户态；

反之，当程序运行在级特权级上时，就可以称之为运行在内核态。

虽然用户态下和内核态下工作的程序有很多差别，但最重要的差别就在于特权级的不同，即权力的不同。 运行在用户态下的程序不能直接访问操作系统内核数据结构和程序。

当我们在系统中执行一个程序时，大部分时间是运行在用户态下的，在其需要操作系统帮助完成某些它没有权力和能力完成的工作时就会切换到内核态。

用户态切换到内核态的**3**种方式

1) 系统调用：这是用户态进程主动要求切换到内核态的一种方式，用户态进程通过系统调用申请使用操作系统提供的服务程序完成工作。而系统调用的机制其核心还是使用了操作系统为用户特别开放的一个中断来实现，例如Linux的int 80h中断。

2) 异常：当CPU在执行运行在用户态下的程序时，发生了某些事先不可知的异常，这时会触发由当前运行进程切换到处理此异常的内核相关程序中，也就转到了内核态，比如缺页异常。

3) 外围设备的中断：当外围设备完成用户请求的操作后，会向CPU发出相应的中断信号，这时CPU会暂停执行下一条即将要执行的指令转而去执行与中断信号对应的处理程序，如果先前执行的指令是用户态下的程序，那么这个转换的过程自然也就发生了由用户态到内核态的切换。比如硬盘读写操作完成，系统会切换到硬盘读写的中断处理程序中执行后续操作等。

六、死锁

死锁是指多个进程在运行过程中，因为争夺资源而造成的一种僵局，如果没有外力推进，处于僵局中的进程就无法继续执行。

死锁原因：

1. 竞争资源：请求同一有限资源的进程数多于可用资源数
2. 进程推进顺序非法：进程执行中，请求和释放资源顺序不合理，如资源等待链

死锁产生的必要条件：

1. 互斥条件:进程对所分配的资源进行排他性的使用
2. 请求和保持条件：进程被阻塞的时候并不释放锁申请到的资源
3. 不可剥夺条件：进程对于已经申请到的资源在使用完成之前不可以被剥夺
4. 环路等待条件：发生死锁的时候存在的一个 进程-资源 环形等待链

死锁处理：

预防死锁：破坏产生死锁的4个必要条件中的一个或者多个；实现起来比较简单，但是如果限制过于严格会降低系统资源利用率以及吞吐量

避免死锁：在资源的动态分配中，防止系统进入不安全状态(可能产生死锁的状态)-如银行家算法

检测死锁：允许系统运行过程中产生死锁，在死锁发生之后，采用一定的算法进行检测，并确定与死锁相关的资源和进程，采取相关方法清除检测到的死锁。实现难度大

解除死锁：与死锁检测配合，将系统从死锁中解脱出来（撤销进程或者剥夺资源）。对检测到的和死锁相关的进程以及资源，通过撤销或者挂起的方式，释放一些资源并将其分配给处于阻塞状态的进程，使其转变为就绪态。实现难度大

七、进程调度算法

先来先服务调度算法**FCFS**： 既可以作为作业调度算法也可以作为进程调度算法；按作业或者进程到达的先后顺序依次调度；因此对于长作业比较有利；

短作业优先调度算法**SJF**： 作业调度算法，算法从就绪队列中选择估计时间最短的作业进行处理，直到得出结果或者无法继续执行；缺点：不利于长作业；未考虑作业的重要性；运行时间是预估的，并不靠谱 ；

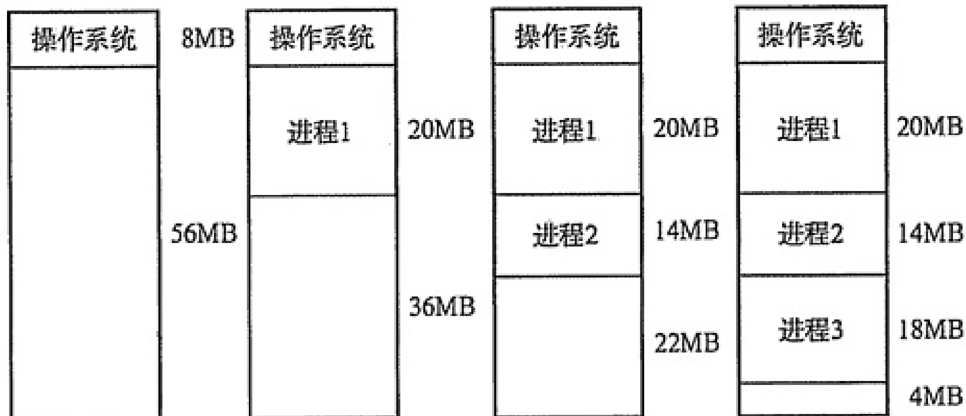
高响应比算法**HRN**： 响应比=(等待时间+要求服务时间)/要求服务时间；

时间片轮转调度**RR**： 按到达的先后对进程放入队列中，然后给队首进程分配CPU时间片，时间片用完之后计时器发出中断，暂停当前进程并将其放到队列尾部，循环 ;

多级反馈队列调度算法： 目前公认较好的调度算法；设置多个就绪队列并为每个队列设置不同的优先级，第一个队列优先级最高，其余依次递减。优先级越高的队列分配的时间片越短，进程到达之后按FCFS放入第一个队列，如果调度执行后没有完成，那么放到第二个队列尾部等待调度，如果第二次调度仍然没有完成，放入第三队列尾部……只有当前一个队列为空的时候才会去调度下一个队列的进程。

一、内存连续分配

主要是指动态分区分配时所采用的几种算法。 动态分区分配又称为可变分区分配，是一种动态划分内存的分区方法。这种分区方法不预先将内存划分，而是在进程装入内存时，根据进程的大小动态地建立分区，并使分区的大小正好适合进程的需要。因此系统中分区的大小和数目是可变的。



首次适应**(First Fit)**算法： 空闲分区以地址递增的次序链接。分配内存时顺序查找，找到大小能满足要求的第一个空闲分区。

最佳适应**(Best Fit)**算法： 空闲分区按容量递增形成分区链，找到第一个能满足要求的空闲分区。

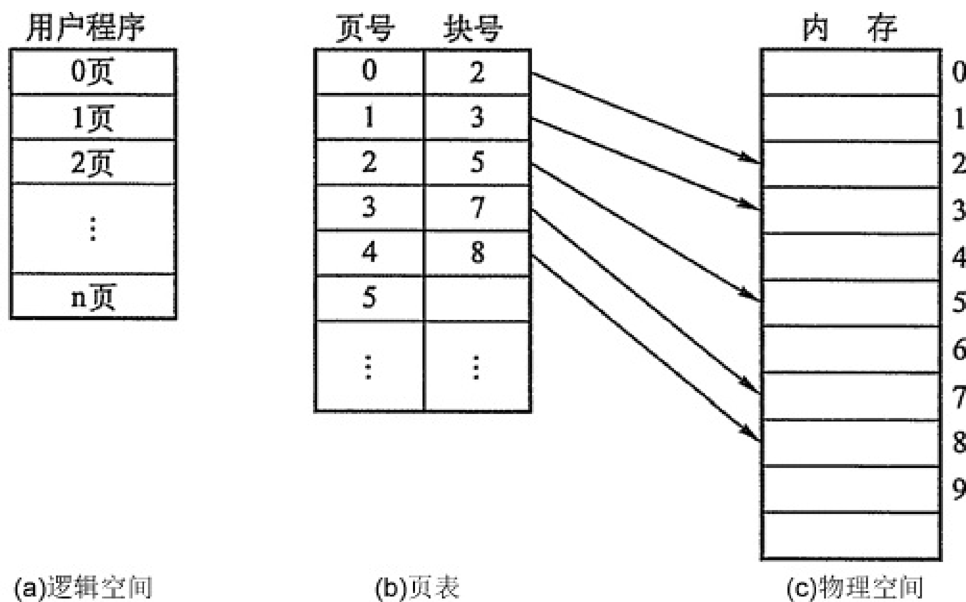
最坏适应**(Worst Fit)**算法： 又称最大适应(Largest Fit)算法，空闲分区以容量递减的次序链接。找到第一个能满足要求的空闲分区，也就是挑选出最大的分区。

二、基本分页储存管理方式

把主存空间划分为大小相等且固定的块，块相对较小，作为主存的基本单位。每个进程也以块为单位进行划分，进程在执行时，以块为单位逐个申请主存中的块空间。

因为程序数据存储在不同的页面中，而页面又离散的分布在内存中，因此需要一个页表来记录逻辑地址和实际存储地址之间的映射关系，以实现从页号到物理块号的映射。

由于页表也是存储在内存中的，因此和不适用分页管理的存储方式相比，访问分页系统中内存数据需要两次的内存访问(一次是从内存中访问页表，从中找到指定的物理块号，加上页内偏移得到实际物理地址；第二次就是根据第一次得到的物理地址访问内存取出数据)。



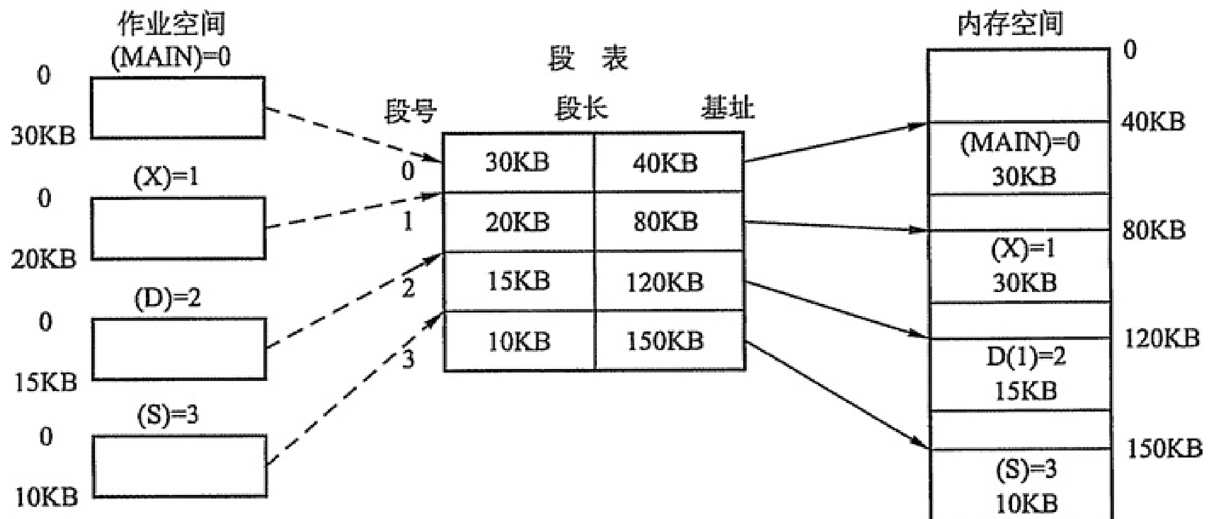
为了减少两次访问内存导致的效率影响，分页管理中引入了快表机制，包含快表机制的内存管理中，当要访问内存数据的时候，首先将页号在快表中查询，如果查找到说明要访问的页表项在快表中，那么直接从快表中读取相应的物理块号；如果没有找到，那么访问内存中的页表，从页表中得到物理地址，同时将页表中的该映射表项添加到快表中(可能存在快表换出算法)。

在某些计算机中如果内存的逻辑地址很大，将会导致程序的页表项会很多，而页表在内存中是连续存放的，所以相应的就需要较大的连续内存空间。为了解决这个问题，可以采用两级页表或者多级页表的方法，其中外层页表一次性调入内存且连续存放，内层页表离散存放。相应的访问内存页表的时候需要一次地址变换，访问逻辑地址对应的物理地址的时候也需要一次地址变换，而且一共需要访问内存3次才可以读取一次数据。

三、基本分段储存管理方式

分页是为了提高内存利用率，而分段是为了满足程序员在编写代码的时候的一些逻辑需求(比如数据共享，数据保护，动态链接等)。

分段内存管理当中，地址是二维的，一维是段号，一维是段内地址；其中每个段的长度是不一样的，而且每个段内部都是从0开始编址的。由于分段管理中，每个段内部是连续内存分配，但是段和段之间是离散分配的，因此也存在一个逻辑地址到物理地址的映射关系，相应的就是段表机制。段表中的每一个表项记录了该段在内存中的起始地址和该段的长度。段表可以放在内存中也可以放在寄存器中。



访问内存的时候根据段号和段表项的长度计算当前访问段在段表中的位置，然后访问段表，得到该段的物理地址，根据该物理地址以及段内偏移量就可以得到需要访问的内存。由于也是两次内存访问，所以分段管理中同样引入了联想寄存器。

分段分页方式的比较

页是信息的物理单位，是出于系统内存利用率的角度提出的离散分配机制；段是信息的逻辑单位，每个段含有一组意义完整的信息，是出于用户角度提出的内存管理机制

页的大小是固定的，由系统决定；段的大小是不确定的，由用户决定

四、虚拟内存

如果存在一个程序，所需内存空间超过了计算机可以提供的实际内存，那么由于该程序无法装入内存所以也就无法运行。单纯的增加物理内存只能解决一部分问题，但是仍然会出现无法装入单个或者无法同时装入多个程序的问题。但是可以从逻辑的角度扩充内存容量，即可解决上述两种问题。

基于局部性原理，在程序装入时，可以将程序的一部分装入内存，而将其余部分留在外存，就可以启动程序执行。在程序执行过程中，当所访问的信息不在内存时，由操作系统将所需要的部分调入内存,然后继续执行程序。另一方面，操作系统将内存中暂时不使用的内容换出到外存上，从而腾出空间存放将要调入内存的信息。这样，系统好像为用户提供了一个比实际内存大得多的存储器，称为虚拟存储器。

虚拟存储器的特征：

1. 多次性：一个作业可以分多次被调入内存。多次性是虚拟存储特有的属性
2. 对换性：作业运行过程中存在换进换出的过程(换出暂时不用的数据换入需要的数据)
3. 虚拟性：虚拟性体现在其从逻辑上扩充了内存的容量(可以运行实际内存需求比物理内存大的应用程序)。虚拟性是虚拟存储器的最重要特征也是其最终目标。虚拟性建立在多次性和对换性的基础上行，多次性和对换性又建立在离散分配的基础上。

五、页面置换算法

最佳置换算法： 只具有理论意义的算法，用来评价其他页面置换算法。置换策略是将当前页面中在未来最长时间内不会被访问的页置换出去。

先进先出置换算法： 简单粗暴的一种置换算法，没有考虑页面访问频率信息。每次淘汰最早调入的页面。

最近最久未使用算法**LRU**： 算法赋予每个页面一个访问字段，用来记录上次页面被访问到现在所经历的时间t，每次置换的时候把t值最大的页面置换出去(实现方面可以采用寄存器或者栈的方式实现)。

时钟算法**clock(**也被称为是最近未使用算法**NRU)**： 页面设置一个访问位，并将页面链接为一个环形队列，页面被访问的时候访问位设置为1。页面置换的时候，如果当前指针所指页面访问为为0，那么置换，否则将其置为0，循环直到遇到一个访问为位0的页面。

改进型**Clock**算法： 在Clock算法的基础上添加一个修改位，替换时根究访问位和修改位综合判断。优先替换访问位和修改位都是0的页面，其次是访问位为0修改位为1的页面。

最少使用算法**LFU**： 设置寄存器记录页面被访问次数，每次置换的时候置换当前访问次数最少的。

1.创建表

语法

```
CREATE TABLE <表名>(<列名> <数据类型>[列级完整性约束条件]
                        [,<列名> <数据类型>[列级完整性约束条件]]…);
```

> 列级完整性约束条件有NULL[可为空]、NOT NULL[不为空]、UNIQUE[唯一]，可以组合使用，但是不能重复和对立关系同时存在。

示例

```
-- 创建学生表
CREATE TABLE Student
(
  Id INT NOT NULL UNIQUE PRIMARY KEY,
  Name VARCHAR(20) NOT NULL,
  Age INT NULL,
  Gender VARCHAR(4) NULL
);
```

2.删除表

语法

```
DROP TABLE <表名>;
```

示例

```
-- 删除学生表
DROP TABLE Student;
```

3.清空表

语法

```
TRUNCATE TABLE <表名>;
```

示例

```
-- 删除学生表
TRUNCATE TABLE Student;
```

4.修改表

语法

```
-- 添加列
ALTER TABLE <表名> [ADD <新列名> <数据类型>[列级完整性约束条件]]
-- 删除列
ALTER TABLE <表名> [DROP COLUMN <列名>]
-- 修改列
ALTER TABLE <表名> [MODIFY COLUMN <列名> <数据类型> [列级完整性约束条件]]
```

示例

```
-- 添加学生表`Phone`列
ALTER TABLE Student ADD Phone VARCHAR(15) NULL;
-- 删除学生表`Phone`列
ALTER TABLE Student DROP COLUMN Phone;
-- 修改学生表`Phone`列
ALTER TABLE Student MODIFY Phone VARCHAR(13) NULL;
```

5.查询

语法

```
SELECT [ALL | DISTINCT] <目标列表达式>[,<目标列表达式>]…
  FROM <表名或视图名>[,<表名或视图名>]…
  [WHERE <条件表达式>]
  [GROUP BY <列名> [HAVING <条件表达式>]]
  [ORDER BY <列名> [ASC|DESC]…]
```

> SQL查询语句的顺序：SELECT、FROM、WHERE、GROUP BY、HAVING、ORDER BY。SELECT、FROM是必须的，HAVING子句只能与GROUP BY搭配使用。

示例

```
SELECT * FROM Student
  WHERE Id>10
  GROUP BY Age HAVING AVG(Age) > 20
  ORDER BY Id DESC
```

6.插入

语法

```
-- 插入不存在的数据
INSERT INTO <表名> [(字段名[,字段名]…)] VALUES (常量[,常量]…);
-- 将查询的数据插入到数据表中
INSERT INTO <表名> [(字段名[,字段名]…)] SELECT 查询语句;
```

示例

```
-- 插入不存在的数据
INSERT INTO Student (Name,Age,Gender) VALUES ('Andy',30,'女');
-- 将查询的数据插入到数据表中
INSERT INTO Student (Name,Age,Gender)
  SELECT Name,Age,Gender FROM Student_T WHERE Id >10;
```

7.更新

语法

```
UPDATE <表名> SET 列名=值表达式[,列名=值表达式…]
  [WHERE 条件表达式]
```

示例

```
-- 将Id在(10,100)的Age加1
UPDATE Student SET Age= Age+1 WHERE Id>10 AND Id<100
```

8.删除

语法

```
DELETE FROM <表名> [WHERE 条件表达式]
```

示例

```
-- 删除Id小于10的数据记录
DELETE FROM Student WHERE Id<10;
```

9.索引

索引是一种特殊的查询表，可以被数据库搜索引擎用来加速数据的检索。简单说来，索引就是指向表中数据的指针。数据库的索引同书籍后面的索引非常相像。

例如，如果想要查阅一本书中与某个特定主题相关的所有页面，你会先去查询索引（索引按照字母表顺序列出了所有主题），然后从索引中找到一页或者多页与该主题相关的页面。

索引能够提高 SELECT 查询和 WHERE 子句的速度，但是却降低了包含 UPDATE 语句或 INSERT 语句的数据输入过程的速度。索引的创建与删除不会对表中的数据产生影响。

创建索引需要使用 CREATE INDEX 语句，该语句允许对索引命名，指定要创建索引的表以及对哪些列进行索引，还可以指定索引按照升序或者降序排列。

同 UNIQUE 约束一样，索引可以是唯一的。这种情况下，索引会阻止列中（或者列的组合，其中某些列有索引）出现重复的条目。

创建索引

语法

```
CREATE [UNIQUE] [CLUSTER] INDEX <索引名> ON <表名>(<列名>[<次序>][,
<列名>[<次序>]]…);
```

> **UNIQUE**：表明此索引的每一个索引值只对应唯一的数据记录 **CLUSTER**：表明建立的索引是聚集索引 次序：可选ASC(升序)或DESC(降序)，默认ASC

示例

```
-- 建立学生表索引：单一字段Id索引倒序
CREATE UNIQUE INDEX INDEX_SId ON Student (Id DESC);
-- 建立学生表索引：多个字段Id、Name索引倒序
CREATE UNIQUE INDEX INDEX_SId_SName ON Student (Id DESC,Name DESC
);
```

删除索引

语法

```
DROP INDEX <索引名>;
```

示例

```
-- 删除学生表索引 INDEX_SId
DROP INDEX INDEX_SId;
```

10.视图

视图无非就是存储在数据库中并具有名字的 SQL 语句，或者说是以预定义的 SQL 查询的形式存在的数据表的成分。

视图可以包含表中的所有列，或者仅包含选定的列。视图可以创建自一个或者多个表，这取决于创建该视图的 SQL 语句的写法。

视图，一种虚拟的表，允许用户执行以下操作：

- 以用户或者某些类型的用户感觉自然或者直观的方式来组织数据；
- 限制对数据的访问，从而使得用户仅能够看到或者修改（某些情况下）他们需要的数据；
- 从多个表中汇总数据，以产生报表。

创建视图

语法

```
CREATE VIEW <视图名>
  AS SELECT 查询子句
  [WITH CHECK OPTION]
```

> 查询子句：子查询可以是任何SELECT语句，但是常不允许含有 `ORDER BY` 子句和 `DISTINCT` 短语；**WITH CHECK OPTION**：表示对UPDATE、INSERT、DELETE操作时要保证更新。

更新视图：

视图可以在特定的情况下更新：

- SELECT 子句不能包含 DISTINCT 关键字
- SELECT 子句不能包含任何汇总函数（summary functions）
- SELECT 子句不能包含任何集合函数（set functions）
- SELECT 子句不能包含任何集合运算符（set operators）
- SELECT 子句不能包含 ORDER BY 子句
- FROM 子句中不能有多个数据表
- WHERE 子句不能包含子查询（subquery）
- 查询语句中不能有 GROUP BY 或者 HAVING
- 计算得出的列不能更新
- 视图必须包含原始数据表中所有的 NOT NULL 列，从而使 INSERT 查询生效。

示例

```
CREATE VIEW VIEW_Stu_Man
AS SELECT * FROM Student WHERE Gender = '男'
WITH CHECK OPTION
```

删除视图

语法

```
DROP VIEW <视图名>;
```

示例

```
DROP VIEW VIEW_Stu_Man;
```

11.ORDER BY

ORDER BY 子句根据一列或者多列的值，按照升序或者降序排列数据。某些数据库默认以升序排列查询结果。

语法

```
    SELECT [ALL | DISTINCT] <目标列表达式>[,<目标列表达式>]…
    FROM <表名或视图名>[,<表名或视图名>]…
    [WHERE <条件表达式>]
    [ORDER BY <列名>] [ASC | DESC];
```

ORDER BY 子句可以同时使用多个列作为排序条件。无论用哪一列作为排序条件，都要确保该列在存在。

示例

```
SELECT * FROM CUSTOMERS
         ORDER BY NAME DESC
```

12.WHERE

WHERE 子句用于有条件地从单个表中取回数据或者将多个表进行合并。

如果条件满足，则查询只返回表中满足条件的值。你可以用 WHERE 子句来过滤查询结果，只获取必要的记录。

WHERE 子句不仅可以用于 SELECT 语句，还可以用于 UPDATE、DELETE 等语句。

语法

```
    SELECT [ALL | DISTINCT] <目标列表达式>[,<目标列表达式>]…
    FROM <表名或视图名>[,<表名或视图名>]…
    WHERE <条件表达式>
```

在指定条件时，可以使用关系运算符和逻辑运算符，例如 `>`、`<`、`=`、`LIKE`、`NOT` 等。

示例

```
SELECT ID, NAME, SALARY
    FROM CUSTOMERS
    WHERE SALARY > 2000;
```

13.LIKE

LIKE 子句通过通配符来将一个值同其他相似的值作比较。可以同 LIKE 运算符一起使用的通配符有两个：

- 百分号（%）
- 下划线（_）

百分号代表零个、一个或者多个字符。下划线则代表单个数字或者字符。两个符号可以一起使用。

语法

% 和 _ 的基本语法如下：

```
SELECT FROM table_name
WHERE column LIKE 'XXXX%'


SELECT FROM table_name
WHERE column LIKE '%XXXX%'


SELECT FROM table_name
WHERE column LIKE 'XXXX_'


SELECT FROM table_name
WHERE column LIKE '_XXXX'


SELECT FROM table_name
WHERE column LIKE '_XXXX_'
```

你可以将多个条件用 AND 或者 OR 连接在一起。这里，XXXX 为任何数字值或者字符串。

示例

下面这些示例中，每个 WHERE 子句都有不同的 LIKE 子句，展示了 % 和 _ 的用法:

语句	描述
WHERE SALARY LIKE '200%'	找出所有 200 打头的值
WHERE SALARY LIKE '%200%'	找出所有含有 200 的值
WHERE SALARY LIKE '_00%'	找出所有第二位和第三位为 0 的值
WHERE SALARY LIKE '2%%'	找出所有以 2 开始，并且长度至少为 3 的值
WHERE SALARY LIKE '%2'	找出所有以 2 结尾的值
WHERE SALARY LIKE '_2%3'	找出所有第二位为 2，并且以3结束的值
WHERE SALARY LIKE '2___3'	找出所有以 2 开头以 3 结束的五位数

14.HAVING

HAVING 子句使你能够指定过滤条件，从而控制查询结果中哪些组可以出现在最终结果里面。

WHERE 子句对被选择的列施加条件，而 HAVING 子句则对 GROUP BY 子句所产生的组施加条件。

语法

下面可以看到 HAVING 子句在 SELECT 查询中的位置：

```
SELECT
FROM
WHERE
GROUP BY
HAVING
ORDER BY
```

在 SELECT 查询中，HAVING 子句必须紧随 GROUP BY 子句，并出现在 ORDER BY 子句（如果有的话）之前。带有 HAVING 子句的 SELECT 语句的语法如下所示：

```
SELECT column1, column2
FROM table1, table2
WHERE [ conditions ]
GROUP BY column1, column2
HAVING [ conditions ]
ORDER BY column1, column2
```

示例

考虑 CUSTOMERS 表，表中的记录如下所示：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

下面是一个有关 HAVING 子句使用的实例，该实例将会筛选出出现次数大于或等于 2 的所有记录。

```
SELECT ID, NAME, AGE, ADDRESS, SALARY
FROM CUSTOMERS
GROUP BY age
HAVING COUNT(age) >= 2;
```

其执行结果如下所示：

```
+----+--------+-----+---------+---------+
| ID | NAME   | AGE | ADDRESS | SALARY  |
+----+--------+-----+---------+---------+
|  2 | Khilan |  25 | Delhi   | 1500.00 |
+----+--------+-----+---------+---------+
```

15.DISTINCT

DISTINCT 关键字同 SELECT 语句一起使用，可以去除所有重复记录，只返回唯一项。

有时候，数据表中可能会有重复的记录。在检索这些记录的时候，应该只取回唯一的记录，而不是重复的。

语法

使用 DISTINCT 关键字去除查询结果中的重复记录的基本语法如下所示：

```
    SELECT DISTINCT column1, column2,.....columnN
    FROM table_name
    WHERE [condition]
```

示例

```
SELECT DISTINCT SALARY FROM CUSTOMERS
         ORDER BY SALARY
```

去除（SALARY 字段）重复记录。

16.AND和OR

AND 和 **OR** 运算符可以将多个条件结合在一起，从而过滤 SQL 语句的返回结果。这两个运算符被称作连接运算符。

AND

语法

```
    SELECT column1, column2, columnN
    FROM table_name
    WHERE [condition1] AND [condition2]...AND [conditionN];
```

将 N 个条件用 AND 运算符结合在一起。对于 SQL 语句要执行的动作来说——无论是事务还是查询，AND 运算符连接的所有条件都必须为 TRUE。

示例

```
SELECT ID, NAME, SALARY
    FROM CUSTOMERS
    WHERE SALARY > 2000 AND age < 25;
```

OR

语法

```
    SELECT column1, column2, columnN
    FROM table_name
    WHERE [condition1] OR [condition2]...OR [conditionN]
```

你可以将 N 个条件用 OR 运算符结合在一起。对于 SQL 语句要执行的动作来说——无论是事务还是查询，OR 运算符连接的所有条件中只需要有一个为 TRUE 即可。

示例

```
SELECT ID, NAME, SALARY
    FROM CUSTOMERS
    WHERE SALARY > 2000 OR age < 25;
```

17.UNION

UNION 子句/运算符用于将两个或者更多的 SELECT 语句的运算结果组合起来。

在使用 UNION 的时候，每个 SELECT 语句必须有相同数量的选中列、相同数量的列表达式、相同的数据类型，并且它们出现的次序要一致，不过长度不一定要相同。

语法

```
    SELECT column1 [, column2 ]
    FROM table1 [, table2 ]
    [WHERE condition]

    UNION

    SELECT column1 [, column2 ]
    FROM table1 [, table2 ]
    [WHERE condition]
```

这里的条件可以是任何根据你的需要而设的条件。

示例

```
SELECT Txn_Date FROM Store_Information
UNION
SELECT Txn_Date FROM Internet_Sales;
```

UNION ALL 子句：

UNION ALL 运算符用于将两个 SELECT 语句的结果组合在一起，重复行也包含在内。

其他类似语句

INTERSECT子句：

用于组合两个 SELECT 语句，但是只返回两个 SELECT 语句的结果中都有的行。

EXCEPT 子句：

组合两个 SELECT 语句，并将第一个 SELECT 语句的结果中存在，但是第二个 SELECT 语句的结果中不存在的行返回。

18.JOIN

连接（**JOIN**） 子句用于将数据库中两个或者两个以上表中的记录组合起来。连接通过共有值将不同表中的字段组合在一起。

考虑下面两个表，（a）CUSTOMERS 表：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

（b）另一个表是 ORDERS 表：

```
+-----+---------------------+-------------+--------+
|OID  | DATE                | CUSTOMER_ID | AMOUNT |
+-----+---------------------+-------------+--------+
| 102 | 2009-10-08 00:00:00 |           3 |   3000 |
| 100 | 2009-10-08 00:00:00 |           3 |   1500 |
| 101 | 2009-11-20 00:00:00 |           2 |   1560 |
| 103 | 2008-05-20 00:00:00 |           4 |   2060 |
+-----+---------------------+-------------+--------+
```

现在，让我们用 SELECT 语句将这个两张表连接（JOIN）在一起：

```
SQL> SELECT ID, NAME, AGE, AMOUNT
        FROM CUSTOMERS, ORDERS
        WHERE  CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

上述语句的运行结果如下所示：

```
+----+----------+-----+--------+
| ID | NAME     | AGE | AMOUNT |
+----+----------+-----+--------+
|  3 | kaushik  |  23 |   3000 |
|  3 | kaushik  |  23 |   1500 |
|  2 | Khilan   |  25 |   1560 |
|  4 | Chaitali |  25 |   2060 |
+----+----------+-----+--------+
```

SQL 连接类型

SQL 中有多种不同的连接：

- 内连接（INNER JOIN）：当两个表中都存在匹配时，才返回行。
- 左连接（LEFT JOIN）：返回左表中的所有行，如果左表中行在右表中没有匹配行，则结果中右表中的列返回空值。
- 右连接（RIGHT JOIN）：恰与左连接相反，返回右表中的所有行，如果右表中行在左表中没有匹配行，则结果中左表中的列返回空值。
- 全连接（FULL JOIN）：返回左表和右表中的所有行。当某行在另一表中没有

匹配行，则另一表中的列返回空值

内连接

语法

```
SELECT table1.column1, table2.column2...
FROM table1
INNER JOIN table2
ON table1.common_field = table2.common_field;
```

示例

考虑如下两个表格，（a）CUSTOMERS 表：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

（b）ORDERS 表：

```
+-----+---------------------+-------------+--------+
| OID | DATE                |          ID | AMOUNT |
+-----+---------------------+-------------+--------+
| 102 | 2009-10-08 00:00:00 |           3 |   3000 |
| 100 | 2009-10-08 00:00:00 |           3 |   1500 |
| 101 | 2009-11-20 00:00:00 |           2 |   1560 |
| 103 | 2008-05-20 00:00:00 |           4 |   2060 |
+-----+---------------------+-------------+--------+
```

现在，让我们用内连接将这两个表连接在一起：

```
    SELECT  ID, NAME, AMOUNT, DATE
     FROM CUSTOMERS
     INNER JOIN ORDERS
     ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

上述语句将会产生如下结果：

```
+----+----------+--------+---------------------+
| ID | NAME     | AMOUNT | DATE                |
+----+----------+--------+---------------------+
|  3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|  3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|  2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|  4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
+----+----------+--------+---------------------+
```

左连接

语法

```
SELECT table1.column1, table2.column2...
FROM table1
LEFT JOIN table2
ON table1.common_field = table2.common_field;
```

这里，给出的条件可以是任何根据你的需要写出的条件。

示例

考虑如下两个表格，（a）CUSTOMERS 表：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

（b）ORDERS 表：

```
+-----+---------------------+-------------+--------+
| OID | DATE                |          ID | AMOUNT |
+-----+---------------------+-------------+--------+
| 102 | 2009-10-08 00:00:00 |           3 |   3000 |
| 100 | 2009-10-08 00:00:00 |           3 |   1500 |
| 101 | 2009-11-20 00:00:00 |           2 |   1560 |
| 103 | 2008-05-20 00:00:00 |           4 |   2060 |
+-----+---------------------+-------------+--------+
```

现在，让我们用左连接将这两个表连接在一起：

```
SELECT  ID, NAME, AMOUNT, DATE
 FROM CUSTOMERS
 LEFT JOIN ORDERS
 ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

上述语句将会产生如下结果：

```
+----+----------+--------+---------------------+
| ID | NAME     | AMOUNT | DATE                |
+----+----------+--------+---------------------+
|  1 | Ramesh   |   NULL | NULL                |
|  2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|  3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|  3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|  4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
|  5 | Hardik   |   NULL | NULL                |
|  6 | Komal    |   NULL | NULL                |
|  7 | Muffy    |   NULL | NULL                |
+----+----------+--------+---------------------+
```

右连接

语法

```
SELECT table1.column1, table2.column2...
FROM table1
RIGHT JOIN table2
ON table1.common_field = table2.common_field;
```

这里，给出的条件可以是任何根据你的需要写出的条件。

示例

考虑如下两个表格，（a）CUSTOMERS 表：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

（b）ORDERS 表：

```
+-----+---------------------+-------------+--------+
| OID | DATE                |          ID | AMOUNT |
+-----+---------------------+-------------+--------+
| 102 | 2009-10-08 00:00:00 |           3 |   3000 |
| 100 | 2009-10-08 00:00:00 |           3 |   1500 |
| 101 | 2009-11-20 00:00:00 |           2 |   1560 |
| 103 | 2008-05-20 00:00:00 |           4 |   2060 |
+-----+---------------------+-------------+--------+
```

现在，让我们用右连接将这两个表连接在一起：

```
SELECT  ID, NAME, AMOUNT, DATE
 FROM CUSTOMERS
 RIGHT JOIN ORDERS
 ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

上述语句将会产生如下结果：

```
+------+----------+--------+---------------------+
| ID   | NAME     | AMOUNT | DATE                |
+------+----------+--------+---------------------+
|    3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|    3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|    2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|    4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
+------+----------+--------+---------------------+
```

全连接

语法

```
SELECT table1.column1, table2.column2...
FROM table1
FULL JOIN table2
ON table1.common_field = table2.common_field;
```

这里，给出的条件可以是任何根据你的需要写出的条件。

示例

考虑如下两个表格，（a）CUSTOMERS 表：

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

（b）ORDERS 表：

```
+-----+---------------------+-------------+--------+
| OID | DATE                |          ID | AMOUNT |
+-----+---------------------+-------------+--------+
| 102 | 2009-10-08 00:00:00 |           3 |   3000 |
| 100 | 2009-10-08 00:00:00 |           3 |   1500 |
| 101 | 2009-11-20 00:00:00 |           2 |   1560 |
| 103 | 2008-05-20 00:00:00 |           4 |   2060 |
+-----+---------------------+-------------+--------+
```

现在让我们用全连接将两个表连接在一起：

```
    SELECT  ID, NAME, AMOUNT, DATE
     FROM CUSTOMERS
     FULL JOIN ORDERS
     ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

上述语句将会产生如下结果：

```
+------+----------+--------+---------------------+
| ID   | NAME     | AMOUNT | DATE                |
+------+----------+--------+---------------------+
|    1 | Ramesh   |   NULL | NULL                |
|    2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|    3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|    3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|    4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
|    5 | Hardik   |   NULL | NULL                |
|    6 | Komal    |   NULL | NULL                |
|    7 | Muffy    |   NULL | NULL                |
|    3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|    3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|    2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|    4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
+------+----------+--------+---------------------+
```

如果你所用的数据库不支持全连接，比如 MySQL，那么你可以使用 UNION ALL子句来将左连接和右连接结果组合在一起：

```
SELECT  ID, NAME, AMOUNT, DATE
  FROM CUSTOMERS
  LEFT JOIN ORDERS
  ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
UNION ALL
  SELECT  ID, NAME, AMOUNT, DATE
  FROM CUSTOMERS
  RIGHT JOIN ORDERS
  ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
```

19.常用函数

COUNT函数是 SQL 中最简单的函数了，对于统计由 SELECT 语句返回的记录非常有用。

要理解 COUNT 函数，请考虑 employee_tbl 表，表中的记录如下所示：

```
SELECT * FROM employee_tbl;
+------+------+------------+--------------------+
| id   | name | work_date  | daily_typing_pages |
+------+------+------------+--------------------+
|    1 | John | 2007-01-24 |                250 |
|    2 | Ram  | 2007-05-27 |                220 |
|    3 | Jack | 2007-05-06 |                170 |
|    3 | Jack | 2007-04-06 |                100 |
|    4 | Jill | 2007-04-06 |                220 |
|    5 | Zara | 2007-06-06 |                300 |
|    5 | Zara | 2007-02-06 |                350 |
+------+------+------------+--------------------+
7 rows in set (0.00 sec)
```

现在，假设你想要统计上表中记录的总数，那么可以依如下所示步骤达到目的：

```
SELECT COUNT(*) FROM employee_tbl ;
+----------+
| COUNT(*) |
+----------+
|        7 |
+----------+
1 row in set (0.01 sec)
```

类似地，如果你想要统计 Zara 的数目，就可以像下面这样：

```
SELECT COUNT(*) FROM employee_tbl
    WHERE name="Zara";
+----------+
| COUNT(*) |
+----------+
|        2 |
+----------+
1 row in set (0.04 sec)
```

注意：所有的 SQL 查询都是不区分大小写的，因此在 WHERE 子句的条件中，ZARA 和 Zara 是没有任何区别的。

CONCAT 函数用于将两个字符串连接为一个字符串，试一下下面这个例子：

```
SELECT CONCAT('FIRST ', 'SECOND');
+----------------------------+
| CONCAT('FIRST ', 'SECOND') |
+----------------------------+
| FIRST SECOND               |
+----------------------------+
1 row in set (0.00 sec)
```

要对 **CONCAT** 函数有更为深入的了解，请考虑 **employee_tbl** 表，表中记录如下所示：

```
SELECT * FROM employee_tbl;
+------+------+------------+--------------------+
| id   | name | work_date  | daily_typing_pages |
+------+------+------------+--------------------+
|    1 | John | 2007-01-24 |                250 |
|    2 | Ram  | 2007-05-27 |                220 |
|    3 | Jack | 2007-05-06 |                170 |
|    3 | Jack | 2007-04-06 |                100 |
|    4 | Jill | 2007-04-06 |                220 |
|    5 | Zara | 2007-06-06 |                300 |
|    5 | Zara | 2007-02-06 |                350 |
+------+------+------------+--------------------+
7 rows in set (0.00 sec)
```

现在，假设你想要将上表中所有的姓名（name）、id和工作日（work_date）连接在一起，那么可以通过如下的命令来达到目的：

```
SELECT CONCAT(id, name, work_date)
    FROM employee_tbl;
+-----------------------------+
| CONCAT(id, name, work_date) |
+-----------------------------+
| 1John2007-01-24             |
| 2Ram2007-05-27              |
| 3Jack2007-05-06             |
| 3Jack2007-04-06             |
| 4Jill2007-04-06             |
| 5Zara2007-06-06             |
| 5Zara2007-02-06             |
+-----------------------------+
7 rows in set (0.00 sec)
```

SUM函数用于找出表中记录在某字段处的总和。

要理解 **SUM** 函数，请考虑 **employee_tbl** 表，表中记录如下所示：

```
SELECT * FROM employee_tbl;
+------+------+------------+--------------------+
| id   | name | work_date  | daily_typing_pages |
+------+------+------------+--------------------+
|    1 | John | 2007-01-24 |                250 |
|    2 | Ram  | 2007-05-27 |                220 |
|    3 | Jack | 2007-05-06 |                170 |
|    3 | Jack | 2007-04-06 |                100 |
|    4 | Jill | 2007-04-06 |                220 |
|    5 | Zara | 2007-06-06 |                300 |
|    5 | Zara | 2007-02-06 |                350 |
+------+------+------------+--------------------+
7 rows in set (0.00 sec)
```

现在，假设你想要获取 daily_typing_pages 的总和，那么你可以用如下命令来达到目的：

```
SELECT SUM(daily_typing_pages)
    FROM employee_tbl;
+-------------------------+
| SUM(daily_typing_pages) |
+-------------------------+
|                    1610 |
+-------------------------+
1 row in set (0.00 sec)
```

你还可以使用 **GROUP BY** 子句来得出不同记录分组的总和。下面的例子将会计算得出每个人的总和，，你将能够得到每个人打的总页数。

```
SELECT name, SUM(daily_typing_pages)
    FROM employee_tbl GROUP BY name;
```